TR05–0007

Final Report – A Programming Logic for Distributed Systems

Mark Bickford and David Guaspari

June 30, 2005

Prepared for:
Dr. Robert Herklotz
Air Force Office of Scientific Research
801 North Randolph Street
Room 732
Arlington, VA 22203-1977

Prepared by:
ATC-NY (Odyssey Research Associates)
Cornell Business & Technology Park
33 Thornwood Drive, Suite 500
Ithaca, NY 14850-1250

Contract No. FA9550-04-C-0106
CRDL No. 0001AC

REPORT DOCUMENTATION PAGE

AFRL-SR-AR-TR-05-

0253

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED Final Report

4. TITLE AND SUBTITLE Scores , A Logical Programming Environment for Distributed Systems	5. FUNDING NUMBERS FA9550-04-C-0106
6. AUTHOR(S) Dr. David Guaspari	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Odyssey Research Associates DBA ATC - NY 33 Thornwood Drive Ithaca, NY 14850-1250	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release, distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

ATC-NY and Cornell University are developing SCorES, a mathematically based tool to support the development of demonstrably correct distributed Systems. SCorES extends to distributed and hybrid systems a paradigm for program development that has been successful in the world of sequential programming-employing methods that are declarative (rather than operational) and constructive. Declarative methods permitsystems to be specified, analyzed, developed, and verified at a conceptual level congenial to human designers. Constructive methods permit automatic code synthesis. Incorporating these methods within the NuPrl environment provides powerful automated support for specifying, developing, verifying, and synthesizing real-time distributed systems at a high level of abstraction.
This report describes two things: a prototype that supports automatic code generation from proofs in a domain-specific logic of distributed systems (one that does not model real-time); an extension of that logic to the domain of hybrid systems, which may contain variables that vary continuously in real time. We demonstrate the code generator by deriving a verifiably correct leader election protocol; and we demonstrate the logic of hybrid systems by applying it to a mutual exclusion algorithm that generalizes Fischer's protocol to distributed systems.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

7-1-05

Abstract

ATC-NY and Cornell University are developing SCorES, a mathematically based tool to support the development of demonstrably correct distributed systems. SCorES extends to distributed and hybrid systems a paradigm for program development that has been successful in the world of sequential programming—employing methods that are *declarative* (rather than operational) and *constructive.* Declarative methods permit systems to be specified, analyzed, developed, and verified at a conceptual level congenial to human designers. Constructive methods permit automatic code synthesis. Incorporating these methods within the NuPrl environment provides powerful automated support for specifying, developing, verifying, and synthesizing real-time distributed systems at a high level of abstraction.

This report describes two things: a prototype that supports automatic code generation from proofs in a domain-specific logic of distributed systems (one that does not model real-time); an extension of that logic to the domain of hybrid systems, which may contain variables that vary continuously in real time. We demonstrate the code generator by deriving a verifiably correct leader election protocol; and we demonstrate the logic of hybrid systems by applying it to a mutual exclusion algorithm that generalizes Fischer's protocol to distributed systems.

Contents

List of Figures

1 Introduction

Distributed systems, now vital to our civilian and military infrastructure, have steadily become more complex—and therefore more difficult to understand, implement, and maintain. As argued in [9], there is no scientific basis for believing that any testing method, or any model of software reliability, can be used to show that a complex software system is "ultrareliable"—e.g., has a failure rate of no more than 10^{-8}/hour. Therefore, if one wishes to make scientific case that a system is ultrareliable, crucial parts of the evidence must come, not from after the fact testing, but from mathematical correctness arguments.

Pioneering work on timed automata and I/O automata (e.g., [35]) has demonstrated the possibility of modeling and analyzing distributed and real-time behaviors mathematically; but great effort is needed to apply current techniques. In our view, these techniques present unnecessary difficulties because they are insufficiently abstract and because they do not formulate the mathematical tasks in their most natural setting. A team at ATC-NY and Cornell University has addressed that problem by extending to distributed and hybrid systems a paradigm for program development that has been successful in the world of sequential programming—employing methods that are *declarative* (rather than operational) and *constructive.*

Declarative methods permit systems to be specified, analyzed, developed, and verified at a conceptual level congenial to human designers. We developed our abstract model, the language of *event structures,* in close collaboration with developers of distributed real-time systems. It can be thought of as a generalization of "message sequence charts." This formalism greatly simplifies the difficult intellectual task of reasoning about complex distributed systems.

Constructive methods permit automatic code synthesis. We can thereby develop a tool, SCorES, that provides powerful automated support for specifying, developing, verifying, and synthesizing real-time distributed systems at a high level of abstraction. In Phase I we have prototyped essential features of SCorES.

The development paradigm that SCorES supports can be characterized as follows:

- Specifications are stated declaratively, in a logical language.

  We use the expressive higher-order logic of NuPrl, in which complex liveness and safety requirements can be stated directly. Declarative specifications are typically much easier to understand—and therefore easier to state correctly—than operational specifications.

- Development steps are inferences in a domain-specific *logic.*

  The domain is that of *event structures*, which formalize and generalize the notion of "message sequence charts" and provide a very natural setting for specification and verification. Using logic as the only development tool simplifies the developer's task: for example, no special machinery is required to carry out *refinement* because, using our methods, refinement is merely logical implication.

- Logical inference is supported by editing, refinement, verification, and information management tools.

  FDL, the prototype formal digital library [13; 6; 5; 4], will provide information management for SCorES, and the NuPrl refiner will provide automated proof support. FDL already includes a large body of useful mathematics and proof tactics developed in NuPrl. In the course of developing components of SCorES we have added libraries of definitions, theorems, and tactics specialized for distributed and hybrid systems.

- Programs are synthesized from proofs and are correct by construction.

  Mathematically speaking, the developer proves that a system with the desired behavior is possible. Such a proof, carried out constructively, implicitly defines a "realizer"—an abstract program that has that behavior. We have prototyped the SCorES component that extracts the realizer and uses it to synthesize a concrete implementation.

Previous work (done jointly at Cornell and ATC-NY) showed how to extend this paradigm to realize those properties of distributed system that make no reference to quantitative real time—for example, typical consensus protocols. The domain of the underlying logic is an abstract representation of distributed execution called an *event structure.* We have used the logic of event structures to specify and derive demonstrably correct distributed algorithms implementing nontrivial mutual exclusion and consensus algorithms (TIP, Peterson, etc.). The specifications are clear and the proofs are the simplest known to us.

The results of this project are:

- A prototype code generator that extracts Java implementations from the realizers on event structures.

  The code generator automatically synthesizes correct implementations of distributed non-real-time specifications. We demonstrate the code generator by generating a verifiably correct implementation of a symmetric leader-election protocol in a ring.

- An extension of the domain of event structures to the domain of *hybrid event structures*, which model systems with real-time behavior, including variables that evolve in continuous time.

  We demonstrate the logic of hybrid event structures by applying it to a mutual exclusion algorithm that generalizes Fischer's protocol to distributed systems.

Section 2 summarizes the starting point of Phase I, the logic of message automata developed in previous work. Section 3 describes the design of the generated code at a high level, and discusses a number of design options that a developer using a full-fledged implementation of SCorES should be able to specify. Section 4 describes the theoretical work of Phase I, extending the logic of message automata to include real-time and time-varying variables. Section 5 summarizes our results.

There are three appendices. Sections A and B provide, respectively, a detailed high level design of the generated code and a listing of the code generated for a leader election protocol. Section C describes a verification, using the SPIN model-checker, of a protocol used by the generated code to suspend threads when they have no useful work to perform.

2 Background: message automata and event structures

This section briefly describes the logic we have defined for distributed systems without quantitative time and without continuous variables. Its basic notions are *event structures* and *message automata.*

An event structure is an abstract algebraic construct that represents one possible execution history of a distributed system. A *specification* of a system asserts that all execution histories of the system satisfy some proposition. We write specifications in the expressive higher-order logic of NuPrl, in which the abstractions that developers use to understand their systems can be stated directly.

The recommended way to use our formalism is to leave the message automata implicit. The user reasons in terms of event structures (message sequence charts) and the system extracts corresponding automata.[1]

Formally, a message automaton is a set of constraints that define a set of event structures—all possible executions consistent with the constraints. The constraints are written in a stylized way so that each one has a clear operational meaning (send a message, update a variable, etc.)

The basic judgment in the logic of message automata, written $A \models \phi$, states that a message automaton A *realizes* a proposition ϕ. This means that

- ϕ is true of every event structure consistent with A; and
- A is *feasible*—that is, the set of event structures consistent with A is nonempty.

We define an operator " $\oplus$ " for composing message automata. If A and B are feasible and *compatible*, then $A \oplus B$ will also be feasible. (Compatibility is a consistency condition briefly discussed below.) Syntactically, composition is the union of all the defining clauses of A and of B; operationally, it means executing A and B concurrently; and logically it corresponds to conjunction.

The clauses of a message automaton are compatible if they satisfy simple consistency conditions that may be checked syntactically. For example, a clause stating that "only action a may change the value of variable x" is *not* compatible with a clause stating that some different action b updates x. Our compatibility rules correspond to compatibility requirements on I/O automata saying that they may be interconnected only by plugging the outputs of one into the inputs of another.

Our domain specific logic of message automata consists of standard logical rules for manipulating formulas, together with logical rules defining the meaning of each of the constructs used to construct message automata.

Base cases: As indicated in section 2.2, a message automaton is a set of clauses that initialize variables, state preconditions for actions, define the effects of actions, etc. Each such clause, on its own, is a message automaton

[1]Because all the underlying machinery is built from NuPrl definitions, those definitions are available to a user who could define a message automaton directly and prove that it satisfies some specification. We don't recommend that, and currently provide no special support for it.

that realizes certain logical formulas, which we'll call *base-case formulas.* These judgments constitute the primitive domain specific rules of the logic. Section 2.3 gives examples of some base-case formulas and their realizers.

Composition: The logical rule associated with composition says that if all the following hold

$$A \text{ and } B \text{ are compatible} \qquad A \models \phi \qquad B \models \psi$$

we may conclude that

$$A \oplus B \models \phi \wedge \psi$$

Development can be carried out by various strategies, but every development can be ordered top-down into a proof that reduces the system specification to some collection of base-case propositions. We can automatically generate realizers for all the base-case propositions and then (if they are mutually compatible) those realizers compose into a message automaton that implements the specification.

2.1 Event structures

Following Lamport, we characterize a run of a program as a set of events. Each event `e` is occurs at a unique *location* `loc(e)`. A location is an abstraction of an *agent* or a *process.* Distinct locations do not share variables; they may communicate with one another only by the events of sending and receiving messages.

Each event has a unique *kind* denoted in one of the following ways

- *receive*(`l`), a receive event on link `l`;
- *local*(`a`), a local event tagged by an identifier `a`.

  Think of `a` as the name of an internal action of an automaton that causes the event.

Instead of introducing a separate notion of a "send" event, we find it technically convenient to say that the effects of any event may include the sending of messages. We insist that if `e` is a receive event, there is a unique event `sender(e)` that sent the message received at `e`. The fully detailed model also stipulates that each message is sent on some specified link.

The events occurring at any one location are totally ordered. We use $e < e'$ to mean that e and e' are events at the same location and e precedes e'; and $e \leq e'$ means, naturally, that $e < e'$ or $e = e'$.

Following Lamport [31], we may define a partial order on the set of *all* events, called the *causal order*. It is the transitive closure of the sender-receiver and predecessor relations. That is, the causal order $\prec$ is the least relation on events such that $\mathtt{e} \prec \mathtt{e}'$ if any of the following is true:

- $\mathtt{e}'$ is a receive event and $\mathtt{e} = \mathtt{sender}(\mathtt{e}')$
- $\mathtt{loc}(\mathtt{e}) = \mathtt{loc}(\mathtt{e}')$ and $\mathtt{e} < \mathtt{e}'$
- for some event $\mathtt{e}''$ we have $\mathtt{e} \prec \mathtt{e}''$ and $\mathtt{e}'' \prec \mathtt{e}'$.

Intuitively, if $\mathtt{e} \prec \mathtt{e}'$, then e is guaranteed to happen before $\mathtt{e}'$; and if e causes $\mathtt{e}'$, then $\mathtt{e} \prec \mathtt{e}'$.

If x and e are associated with the same location, "x when e" denotes the value of the state variable x immediately before event e, and "x after e " denotes its value immediately after e. (If they are not associated with the same location, the two phrases are meaningless.)

Every event e also has a *value* val(e). The value of a receive event is the message that is received. The value of a local event is a modeling artifact that may be defined however we like.

In an event structure, the collection of events, locations, and variables must satisfy a few simple axioms, including:

1. An event may send only a finite number of messages.
2. At any location, every event but the first has a unique predecessor event (at that location). The predecessor of e is called pred(e).
3. The causal order is well-founded.
4. The observable value of a state variable is changed only by the occurrence of an event (at its location). That is,

   $$\mathtt{x\ when\ e} = \mathtt{x\ after\ pred(e)}$$

   which says that x cannot change in between events.

A powerful higher-order logic allows us to define and make use of arbitrarily complex derived concepts on top of the event structure model. For example, we can define history operators that list or count previous events having certain properties. We can also define useful notations for concepts such as "event e changes state variable x."

2.2 Message Automata

Event structures are infinite objects, but they arise from the behaviors of *finite* distributed programs. We call our representations of these finite programs *message-automata.* A message-automaton is a finite collection of declarations and clauses each of which is assigned to a unique location. The *declarations* provide the names and types of state variables, local actions, input messages, and output messages. The *clauses* defining an automaton

- *initialize* variables
- define local actions of the automaton by
  - *stating preconditions* for an action
  - defining its *effects* on state variables
  - defining any *output* messages the action generates
- state *frame conditions* stipulating which actions may change a given state variable or send messages on a given link

In general, a precondition is a predicate `P(state,v)`. (Its parameters always have the name `state` and `v`.) It is *satisfied* if, giving `state` the value of the current state, there is *some* value of `v` such that `P(state,v)` is true. An action is eligible iff its precondition is true; and executing an eligible action means nondeterministically assigning some satisfying value to v, which becomes the value of the action, and then performing a change of state and/or sending messages. The change of state and the messages to be sent may be defined parametrically in terms of the action's value.[2]

The message-automaton shown in figure 1 was extracted from the proof of the trivial two-phase handshake protocol that will be described in section 2.4. The meaning of the mnemonically-named constructs ought to be reasonably clear.

[2]In future versions of the semantics we plan to make a subtle change in these semantics—from making a nondeterministic choice to making a random choice.

```
action $a$; precondition $ready = true$
     effect $ready := false$
     sends $[< out, x >]$
only $\{a, rcv(in)\}$ affects $ready$
only $a$ sends $out$
```

Figure 1: A simple message automaton

2.3 Base-case rules

Message automata contain only six kinds of clauses, and each gives rise to one base-case (or primitive) logical rule. Here are instances of two of them. We abbreviate "message automaton M realizes the formula ϕ" by the expression "$M \models \phi$.

Initialization clause

$$@i\ \textbf{state}\ x : T; \textbf{initially}\ x = v \quad \models \quad \forall e@i.\ \mathit{first}(e) \ \Rightarrow\ x\ \textbf{when}\ e \ = \ v$$

An automaton that initializes the local variable x of location i to value v realizes the following proposition: if e is the first event at i, then the value of (x **when** e) is v. The phrase "$\forall e@i.$ " means "for every event e at location i."

Effects clause

$$\begin{array}{ll} @i & \textbf{state}\ x : T1; \\ & \textbf{action}\ k : T2; \\ & k(v)\ \textbf{effect}\ x := f(s, v)\ \models \\ & \forall e@i.\ kind(e) = k\ \Rightarrow\ x\ \textbf{after}\ e\ =\ f(state\ \textbf{when}\ e, val(e)) \end{array}$$

Every effect clause has two implicit bound variables: v, denoting the value of the action; and *state*, denoting the current state. The effects clause above says that one effect of action k is to update x, as a function of the value of k. This clause realizes the proposition that after any event e of kind k, x has the specified value.

2.4 Example: two-phase handshake

We use a very simple example to illustrate how a specification is expressed naturally as a proposition on event structures and how it is refined *entirely in the logical domain* to a collection of simpler specifications that can be directly realized using the primitive rules. The specification is parameterized by a pair of links `in` and `out` such that the destination of `in` is the source of `out`. The informal specification is:

> Between any two sends on link `out` there is a receive on link `in`.

We define two convenient predicates:

- $R(e)$ means that event e is a receive event on link `in`.
- $S(e)$ means that event e is a send event on link `out`. (More formally: there is a receive event e' on link `out` such that e is its associated sender.)

Formalizing these definitions is straightforward but would require us to introduce otherwise unnecessary technical detail about event structures.

In these terms, our specification can be formalized as

$$\forall e_1, e_2.\ (e_1 < e_2 \ \wedge\ S(e_1)\ \wedge\ S(e_2)) \ \Rightarrow\ \exists e'.\ e_1 < e' < e_2 \ \wedge\ R(e') \qquad (1)$$

This formula does not mention any local action or local state variable. Those are implementation details, not specification concepts.

The first reduction step introduces an implementation decision: a single local action a will do all the sending. To express this we first define a predicate *snd*:

- $snd(e)$ means that e is an event whose kind is $local(a)$

A system in which action a does all the sending satisfies proposition (2):

$$\forall e.\ S(e) \ \Leftrightarrow\ snd(e) \qquad (2)$$

To realize proposition (2) it suffices to realize

$$\forall e. snd(e) \ \Rightarrow\ S(e) \qquad (3)$$
$$\forall e. S(e) \ \Rightarrow\ snd(e) \qquad (4)$$

But we can show that

$$\begin{array}{rl} \texttt{local(a)}\ \textbf{sends}\ [x]\ \textbf{on}\ \mathit{out} & \models\ (3) \\ \textbf{only}\ \texttt{local(a)}\ \textbf{sends on}\ \mathit{out} & \models\ (4) \end{array}$$

The proofs are similar. One can formally prove (3) by appealing to the base-case rule saying that the clause "local(a) **sends** $[x]$ **on** *out*" realizes the proposition

$$\begin{array}{l} \forall e@src(out).\ kind(e) = local(a)\ \Rightarrow \\ \qquad \exists e'.kind(e') = rcv(out)\ \wedge\ sender(e') = e\ \wedge\ val(e') = x \end{array} \tag{5}$$

and then showing that (5) logically implies (3). A user following recommended practice simply shows that (5) implies (3) and does nothing further with (5), letting it remain as a leaf of the proof tree. We provide a tactic, `CreateRealizer`, whose effects include recognizing (5) as the conclusion of a base-case rule, creating the corresponding clause, and adding that clause to the definition of the realizer that the proof will ultimately create.

Given (2), the specification (1) becomes equivalent to

$$\begin{array}{l} \forall e_1, e_2.\ \Rightarrow\ (e_1 < e_2\ \wedge\ snd(e_1)\ \wedge\ snd(e_2)) \\ \qquad \exists e'.\ e_1 < e' < e_2\ \wedge\ R(e') \end{array} \tag{6}$$

The next reduction step introduces another implementation decision: a local state variable, *ready*, will control the sending. It must be true when a send occurs, and be set to false by performing the send. That decision is expressed by the proposition

$$\forall e.\ snd(e)\ \Rightarrow\ (ready\ \textbf{when}\ e)\ \wedge\ \neg(ready\ \textbf{after}\ e) \tag{7}$$

We can realize (7) by a message automaton defined as follows:

precondition on a : ready = true
effect local(a) : ready := false

A fully formalized argument would reduce (7) to base-case rules and ordinary logic, as was done for proposition (2).

So, given (2) and (7), the specification becomes equivalent to

$$\begin{array}{l} \forall e_1, e_2.\ e_1 \le e_2\ \wedge\ \neg(ready\ \textbf{after}\ e_1)\ \wedge\ ready\ \textbf{when}\ e_2\ \Rightarrow \\ \qquad\qquad \exists e'.\ e_1 < e' < e_2\ \wedge R(e') \end{array} \tag{8}$$

Finally, we insist that *ready* is changed only by send or receive events, that is:

$$\forall e.\ ready\ \Delta\ e\ \Rightarrow\ snd(e)\ \vee\ R(e) \tag{9}$$

The expression *ready* Δ *e* formalizes the notion that "event e changes *ready*."

Proposition (9) is realized by the following message automaton:

$$\textbf{only}\ [\texttt{local(a)}, \texttt{rcv(in)}]\ \textbf{affect} = \texttt{ready}$$

We have now derived an implementation, because the propositions (2), (7), and (9) together imply (1), and because the realizers we have used for those propositions are compatible. Equivalently, we may show that proposition (9) implies proposition (8). This implication follows from a general lemma saying that if the value of a variable changes between events e_1 and e_2, then some event that changes it must occur in between e_1 and e_2. This lemma holds for all event structures, and can be formally proven by induction over the well-founded causal order.

Gathering these clauses together gives the automaton shown in figure 1.

Suppose we subsequently choose to strengthen the specification (1) by adding a liveness condition: "after every receive on in there will be a send on out." The strengthened specification follows from (2), (7), and (9) together with the additional property:

$$\forall e.\ kind(e) = rcv(in)\ \Rightarrow\ ready\ \textbf{after}\ e$$

So all we need do is derive a realizer for this proposition that is compatible with the one already derived.

2.5 Event Structure Patterns

As illustrated in the simple sender/receiver example, users of our method refine specifications by logical implication until they are realizable by message automata. It would be onerous to require the user to reduce propositions all the way down to base-case formulas.

Each base-case proposition corresponds to a simple, low-level programming construct, a clause in the definition of a message automaton. One way to ease the user's task is to introduce rules for higher-level programming constructs. An analogy with the logic of sequential programming languages may be helpful: In a rule of the form "$M \models \phi$", the proposition ϕ should, intuitively, correspond to the strongest postcondition of M.

To illustrate this idea we have defined a notation for describing *event structure patterns* (ESP), inspired by the Event Correlation Language defined at Stanford. Similar languages form parts of other distributed program specification languages. We may think of ESP expressions as simple programming patterns.

To illustrate the use of ESP, suppose that at some point in the derivation of a program, we find that we must realize the proposition:

$$\begin{array}{l}\text{Every event of kind } A \text{ will send a message on link } l,\\ \text{provided no events of kind } B \text{ have previously occurred.}\end{array} \tag{10}$$

Implicitly, we are referring only to events at a single location. All ESP expressions will define specifications that are local, in this sense.

In general, ESP expressions will allow us to say that event e sends a certain kind of message (or updates a state variable) if and only if the events up to and including e match some pattern. Propositions such as this are basic building blocks of many distributed algorithms; and, if the pattern is suitably simple, they are clearly realizable—by a finite state machine that on every event checks whether it "completes the pattern" and, if so, takes an appropriate actions.

Syntactically, an ESP expression E is formed by combining "basic" ESP expressions with operators reminiscent of, but richer than, regular expression operators. The meaning of a regular expression is the set of strings that match it (the *language it accepts*). Similarly, we will say that an ESP expression *accepts* the interval $[e_1, e_2]$ when the list of events between e_1 and e_2 (and their values and local states) match the pattern defined by the ESP expression. Each ESP expression is assigned to some location i and accepts only intervals of events occurring at that location. Since we are usually interested in the case in which e_1 is the first event at that location, we will say that an ESP expression accepts event e if it accepts the interval $[\mathit{first}(i), e]$ where $\mathit{first}(i)$ is the first event at location $i = \mathit{loc}(e)$.

Accordingly, we may introduce infinitely many new base-case proof rules

$$E \models \phi_E$$

where E is an ESP expression and ϕ_E, a logical formula defined by recursion on the construction of E, expresses its declarative meaning. We also define a message automaton M_E that realizes ϕ_E. Strictly speaking, $E \models \phi_E$ is a shorthand for $M_E \models \phi_E$. To repeat the analogy with sequential programming, ϕ_E is something like the strongest postcondition of M_E.

In logical terms, these new rules can be thought of as derived rules of inference. They do not increase the formal power of the logic, but simply permit the direct application of frequently occurring proof patterns. For the user, this effects a great simplification: The construction of M_E and the proof that M_E realizes ϕ_E is, naturally, an elaborate induction. But it is done once and for all, and built into the logic. So the user does *not* have to do it—in particular, does not have to construct trivial but tedious induction invariants, etc.

We now give a brief description of ESP. A basic ESP expression is a pair $(k, test)$ where k is an event kind and *test* is a predicate expressed in terms of the event's value and the local state when the event occurs. So, for example, if *uid* is a local variable, then each of the following is a basic ESP expression:

$$\begin{aligned} R_1 &= (rcv(vote), val > uid) \\ R_2 &= (rcv(vote), true) \end{aligned}$$

The basic expression $(k, test)$ accepts e if e is the first event of kind k whose state and value satisfy *test*. Thus, R_1 accepts the first *vote* message received in state in which $val > uid$; and R_2 accepts the first *vote* message received.

If E and F are ESP expressions, then so are

- $(E|F)$, which accepts the first event accepted by either E or F.
- $(E\&F)$, which accepts the first e for which both E and F have accepted some $e' \leq e$.
- $(E;F)$, which accepts e if for some e', E accepts $pred(e')$ and F accepts $[e', e]$.

The explanation of ";" makes it clear that the explanations just given for "|" and "&" have slightly cheated: for we must define what it means to accept an *interval* of events, and not merely what it means to accept a single event. A proper account is straightforward, but contains distracting details.

To see why ESP must be more general than regular expressions consider the negation operator. Regular expressions are defined with respect to a fixed alphabet like $\{a, b, c, d\}$; and therefore an expression like $\neg a$ is simply a shorthand for $(b|c|d)$.

But we wish to use ESP expressions without knowing the relevant "alphabet" in advance. Another way to say that is that, if E is an ESP expression, we want to be able to compose M_E with any other compatible automaton, which may have an arbitrary set of actions. Since we can't take the pattern for "an A event preceded by any non-B events" to be the regular expression $(\neg B)^*A$, we need another kind of control mechanism.

Instead of a negation operator we introduce *exceptions* (denoted by natural numbers) and generalize the notion of *accepts* to the notion "E accepts$_n$ $[e_1, e_2]$"—where the subscript stands for the exception status. The status $n = 0$ is *normal* and all $n > 0$ are *exceptional.* We add throw (T), catch (C), and loop $(*)$ operators. If E is an ESP expressions, then so are

- $T_n(E)$, which accepts$_n$ e when E accepts$_0$ e or E accepts$_n$ e.
- $C_n(E)$, which accepts$_0$ e when E accepts$_0$ e or E accepts$_n$ e.
- $[E]*$, which accepts$_n$ $[e_1, e_2]$ only when $n > 0$ and for some $a_0, \ldots, a_k$, $a_0 = e_1$ and E accepts$_0$ $[a_i, pred(a_{i+1})]$ and E accepts$_n$ $[a_k, e_2]$.

The final ingredient of ESP is a way to mark an expression with an action—either to send a message or to update a state variable. The syntax is:

- $E.send_l(s, v.t)$, where t is a function of the state s and event value v.

  Whenever E accepts e, e sends $t($state **when** $e, val(e))$ on link l.

- $E.x := s, v.t$, where t is a function of the state s and event value v.

  Whenever E accepts e, x after e $= t($state **when** $e, val(e)$.

An action mark does not affect which events are accepted by the expression, but instead defines an *action assertion*: every event that is accepted normally by the subexpression up to the action mark will perform the indicated action and these actions are preformed only by events that the subexpression accepts normally. (We will illustrate this more precisely in section 2.6.) The proposition ϕ_E is the conjunction of all the action assertions defined by E. As already noted, we have shown that every ESP expression is realizable—by giving a constructive proof that essentially *compiles* an ESP expression to a message automaton.

A final note on the limitations of ESP: ESP expressions define purely reactive programs. They don't initiate events, but only react to them. So the message automata constructed from them contain only init, effect, sends, and frame clauses and do not include any precondition clauses.

2.6 ESP Example

To restate the point of ESP: to allow the developer to use a simple program pattern as a realizer of some relatively low-level proposition. Under the hood, our tools derive a logical proposition that is declarative specification for that program pattern to apply in the reasoning steps and a message automaton that realizes that proposition.

```
Loop{ if an $A$ event occurs then send $x$ on link $l$;
      if a $B$ event occurs then exit the loop;}
```

This description is slightly ambiguous, because it is not clear whether events of any other kind can send on link l. We resolve the ambiguity by saying that they cannot.

The informal operational description translates straightforwardly to an ESP expression. If A is an event kind, we use A as a shorthand for the basic ESP expression $(A, true)$, which accepts the first event of kind A that it sees. Using that shorthand, we can represent our operational description with the ESP expression

$$E \equiv [A.send_l(x)|T_1(B)]* \tag{11}$$

The loop performs a send action on every A event and exits, by throwing exception 1, on any B event.

Intuitively, we might state the declarative meaning of E, its strongest postcondition, as "an event sends on link l iff it has kind A and there have been no prior events of kind B; and the message it sends on l is x." Call this proposition ϕ'. The generated specification, ϕ_E, should be equivalent to ϕ' but may not be as intuitive one or as easy to apply in a proof. We now sketch, by example, two things: how we generate ϕ_E and how ϕ_E can be automatically simplified.

Generating ϕ_E

A accepts$_0$ $[x,y]$ translates to

$$\phi_1 \equiv \text{“}y \text{ is the first event in } [x,y] \text{ of kind } A\text{”}$$

$(A|T_1(B))$ accepts$_0$ $[x,y]$ translates to

$$\phi_2 \equiv \phi_1 \wedge \text{“no event in } [x,y] \text{ other than } y \text{ is of kind } B$$

Now consider iteration. If $Q(x,y)$ is a predicate defining any relation between events, we may define a predicate Q^+, on event intervals, as follows:

$$Q^+[e_1,e_2] \Leftrightarrow \exists a_0,\ldots,a_k.\forall i < k.Q(a_i, \mathit{pred}(a_{i+1})) \ \wedge \ a_0 = e_1 \ \wedge \ Q(a_k,e_2)$$

In other words, Q^+ is true over an interval if the interval can be decomposed into one or more subintervals over which Q holds.

Then the action assertion for $[A.\mathit{send}_l(x)|T_1(B)]*$ is that e sends on l if and only if

$$[A|T_1(B)]*; (A|T_1(B)) \text{ accepts}_0 \ e$$

which translates to

$$(\phi_2)^+[\mathit{first}(e), e]$$

The full definition of the translation, though straightforward, is complex.

Simplifying ϕ_E and finding invariants

The point of ESP will be lost if ϕ_E is difficult for the user to understand or awkward to apply in a proof. The output of the translation is highly stylized, and a number of useful simplifications can be stated as rewrite rules, for example, if $P(x,y)$ is the predicate $\forall e \in [x,y].\ \phi$, then

$$P^+[e_1,e_2] \Leftrightarrow \forall e \in [e_1,e_2].\ \phi$$

That is: an invariant may be established over an interval by decomposing the interval into subintervals all of which satisfy the invariant. More powerful induction principles can also be stated as rewrites.

Automatically applied rewrite rules should be able to simplify the ϕ_E generated for example 11 to something very close to ϕ'. One of our research tasks in Phase II will be to define a set of rewrites and proof tactics adequate to this purpose in typical cases. We claim that our method will not need automated invariant discovery tools since most invariants these tools find are implicit in the constructive proof of the realizers for ESP expressions or follow from the general rewrite rules about their meanings.

3 Generating code from message automata

Section 2 has indicated how our methods proceed, in the non-real-time case, from a specification to a message automaton that provably implements the specification.[3] It also notes some of the Phase II research tasks necessary to realize that body of work in a practical tool.

This section presents a high-level description of the prototype code generator developed in Phase I. Input to code generation is a table that assigns to each location either an IP address or an indication that the location is "external." The locations that are not external are "internal."

Our code defines the behavior at internal locations. An external location, by contrast, might represent a device. The correctness proof for the code we generate might therefore rely on explicit hypotheses about the behavior of external nodes. As will be seen, the prototype code generator creates dummy implementations for the external nodes, so that messages from an external location can be simulated by entering them through a gui, and messages sent to an external location are simply marshaled and displayed on a terminal.

The output of the code generator is a set of Java programs, one associated with each location, that collectively implement the specification. This section gives an overview of the generated code (section 3.1) and of how it's generated (section 3.2). It also briefly mentions some foundational questions that are outside the scope of this work (section 3.3).

The Phase I prototype applies to a static situation, in which all locations and their links are known in advance. That is because the formal model uses a standard mathematical trick to represent dynamic situations as static: All possible nodes and all possible links between them exist already. Dynamic behavior, such as the creation of a node, can be mathematically simulated by introducing some "activation" event that starts the preexisting node running. Generating code that doesn't, for example, merely simulate the dynamic creation of a link but actually creates one will require us to make those idioms for mathematical simulation completely systematic, so that the code generator can recognize them. That is a subject for Phase II.

[3]Section 4.4 applies the logic of message automata to derive (and automatically generate code for) a symmetric leader election protocol.

3.1 Design of the generated code

The generated code is very straightforward. The main elements of our abstract model are locations (which can be thought of as computational nodes) and links (one-way FIFO communication channels between locations). We generate one Java package for each location and one for each link.

The package for a link defines the types that can be sent across the link, interfaces `Sender` and `Receiver` for using the link, and a correlated pair of classes that provide default implementations of this interfaces sending. For a link implemented as a socket connection, these correlated classes agree on the IP address and port number to which messages are being sent.

The package for each location defines a class `Node` that encapsulates the behavior of the automaton at that location, and a class `Main` with a main program that creates the node and sets it running. Creating the node means, essentially, supplying it with `Sender` and `Receiver` objects for the links to which it is attached.

This simple organization localizes the changes that have to be made in order to use a different implementation of a link, or replace our dummy version of an external node with a simulation of it, or with the real thing.

The prototype makes, by default, a number of implementation decisions that do not affect the correctness of the implementation but may affect its performance. A real-world tool must allow developers to make those decisions in order to tune their systems. In a fully developed tool, these decisions would be supplied to the code generator as additional parameters.

We describe the default implementation, and list some of the important parameters that a Phase II implementation would provide.

3.1.1 Links

A link, providing one-directional communication, is implemented by a send object and a receive object. The Java code for a link defines a `Receiver` interface, a `Sender` interface, and default implementations of them.

Our default implementation of a link between internal nodes is a socket connection. The receiver creates a server socket and a new thread that listens on the socket and inserts every message that arrives into an internal buffer. The receiver exports methods for inspecting the buffer and for extracting its elements. The socket in the sender knows the IP address and port on which the receiver listens and exports methods for serializing and sending messages across the socket.

Here are some of the implementation decisions about links that a user should be able to control:

Initialization Even if we are implementing a system that takes failure into account, our formal model assumes an initial state consisting of some set of functioning nodes and links. The first thing the generated code does is to set up all those links, assuming that the nodes they connect are initially available. If any node is unavailable, our setup procedure will fail. A user may want to use a different initialization protocol, or to define a recovery mechanism for this initialization procedure.

Security and integrity In our default implementation, the receiver listens on a predetermined port. Our semantic model assumes that only the appropriate sender object sends to that port. A user will want to control how that assumption is enforced, which will require a realistic evaluation of threats: Are we worried about misconfiguration or malice? An insider or an outsider? Does it suffice to verify the identity of the machine that sends a message (which can be done by IPSEC) or must one identify the process that sends it? Etc.

Flow control Our formal model requires that the messages on any one link are transmitted reliably, and in order. The prototype accomplishes this by "pushing" messages from the sender and buffering them at the receiver. A developer may prefer a different model—for example, communication by "pull": messages are buffered at the sender and transmitted only when the receiver signals its readiness.

Optimizations Many simple optimizations are possible. For example, if two locations reside on the same host, they may communicate directly through a shared buffer, without setting up a socket.

3.1.2 Nodes

Each node has a repertoire of actions, and each action guarded by some trigger or condition (the receipt of a message and/or a predicate on its local state). The behavior of a node is, repeatedly, to choose some eligible action and perform it, thereby updating its state and/or sending messages.

Here are some of the implementation decisions about nodes that a user should be able to control:

Scheduling Our formal semantics requires that actions be chosen fairly: that is, an action cannot become permanently eligible and yet be permanently ignored. Our default implementation of fairness is round-robin. In order to tune the performance of the system, or to create a system that adapts its behavior to conditions, a user may employ any scheduling discipline (any scheme of priorities, timeouts, etc.) that selects actions fairly.

External nodes In our formalism, all actors are message automata residing at particular locations. When those actors are software, we generate the code. But some actors will be, for example, I/O devices. We cannot generate the code guaranteeing that an input or output node behaves correctly, but must simply assume that it does so. (We may, of course, formally specify and implement defensive behavior that achieves the desired result if an external node behaves correctly, and acts appropriately if not.) The prototype provides a virtual model of external nodes.

A reasonable virtual model (of which the prototype implements a simplified version) would work as follows: Every external node is a gui. The gui has a separate panel for each link end to which it is connected: for entering input when it can send to a link, and displaying inputs when it can receive from one.

To limit the amount of gui programming required, we restrict external nodes as follows: each is connected to only one link end. Messages received by an external node are written to the screen. If an external node can send to a link, we create a gui for entering the messages and sending them.

3.2 Implementation of the code generator

3.2.1 Scope

In our formal model, the internal state variables of a message automaton and the messages transmitted between them may have any type definable in NuPrl. The prototype limits messages to the "basic" types of integer, boolean, and string; and it limits state variables to types constructed from basic types by the product and union operations (in C terms, to structs composed from basic types). The Phase I prototype covers any automata generated from an ESP expression.

3.2.2 Code generation

Intuitively, a constructive proof of a proposition defines a realizer for the proposition. For example, a realizer of the proposition that every pair of integers has a greatest common divisor is a (demonstrably correct) algorithm for a function that computes it.

Our logic for distributed systems is defined so that the realizer of a property—e.g., mutual exclusion—is a message automaton that implements the property. In this logic, a proof consists of three parts:

- a message automaton (constructed automatically by the NuPrl environment),
- a proof that the automaton is *feasible* (roughly speaking, implementable), and
- a proof that all executions of the automaton satisfy some given set of requirements.

Thus the proof contains both a recipe for how to behave (the automaton and elements of the feasibility proof) and a demonstration that the behavior implements some desired goal.

The realizer is a NuPrl term. The code generator walks through the term until it reaches primitives, whose translation it looks up in a table and inserts appropriately into the target text. This makes it easy to extend the code generator by adding new "primitive" symbols to the lookup table.

Code generation uses the feasibility proof as follows: To implement the logic that decides when an action is eligible we need an algorithm for evaluating its precondition. The feasibility proof requires us to show, constructively, that all such preconditions are decidable, and from that we automatically extract algorithms to carry out all the decisions.

3.3 Foundations

We note a foundational question that is outside the scope of this project. Underlying NuPrl is a version of lambda calculus, and each NuPrl term has an executable meaning defined by applying reduction rules. Therefore, every NuPrl terms is a program. In the "classic" proofs as programs method, the realizer of a proof is therefore a program that implements the proposition proved.

In our logic for distributed systems a realizer is a message automaton, which is a NuPrl term, but execution of *that* NuPrl term does not implement the propositions that the message automaton realizes. (That's why we need a separate code generation step.) The reason is that a message automaton is really a predicate that defines a set of execution histories. The corresponding NuPrl term is therefore a program that implements that predicate: a higher-order program that recognizes whether a given execution history satisfies the constraints. It is not a distributed program that actually implements appropriate execution histories.

4 A logic for hybrid systems

In Phase I we have defined a semantics and a logic for *hybrid* message automata, which incorporate real time and variables that may vary continuously in real time. We do so by adapting modeling principles that have been successfully used in other formalisms for hybrid systems: In these models the history of a hybrid system may contain discrete transitions, taking zero time, that update the values of discrete variables and—speaking in the jargon of control theory—selects the control laws constraining the evolution of continuous variables. In the intervals between discrete transitions, continuous variables evolve according to constraints of the current control laws.

4.1 Hybrid event structures

Formally, we may choose to model time by any subgroup `Time` of $(Real, +)$. Choosing the "right" time domain will require more experience. We currently use the domain of rational numbers, because we want a model of time that is *dense*—between any two instants of time there are infinitely many other instants—and because the rationals are easier to model in NuPrl than the real numbers. (Note that any time value occurring in a program, such as the value returned by a call to a clock, will be a rational number.)

We assume a conceptual global clock.[4] In physical terms, this means that

[4]There is no problem in modeling the possibility that local clocks consulted at particular

relativistic effects can be ignored. We add to language of an event structure a function *time* so that *time*e is the value of the global clock at event e; and define d to be the corresponding metric

$$d(\mathtt{a}, \mathtt{b}) = |time(\mathtt{b}) - time(\mathtt{a})|.$$

All variables now represent trajectories. The denotation of a variable x of type T is a function $[0, +\infty) \longrightarrow T$. To say that f is the denotation of x "now" is to say that $f(t)$ will be the value of x after time t has elapsed (from "now") if no event occurs to change it: the value of a variable at location i can be changed only by an event at location i.

An event may cause a discontinuity, and may also change the "rule" by which a variable evolves. To express these principles we merely generalize the `when-after` axiom (axiom 4, of section 2.1): For all $t \geq 0$,

$$\neg first(\mathtt{e}) \;\Rightarrow\; (\mathtt{x\ when\ e})(t) = (\mathtt{x\ after\ pred(e)})(t + d(\mathtt{pred(e)}, \mathtt{e}))$$

A discrete variable is one whose denotation is required to be a constant function. For discrete variables, this axiom is equivalent to the previous one.

This small repertoire of basic operations allows us to introduce arbitrarily complex defined operators—for example, an operator that stitches together the sequence of denotations of a trajectory variable x into a single function $\tau_{\mathtt{x}} : [0, +\infty) \longrightarrow T$ representing its entire history, parameterized by the global clock. At any time t, $\tau_{\mathtt{x}}(t)$ is the value of variable x.

Suppose, for example, that all the following hold:

e1 and e2 are successive events at location i
$d(\mathtt{e1}, \mathtt{e2}) = 2$
$(\mathtt{x\ after\ e1}) = (\mathtt{x\ when\ e2}) = \lambda t.\ t$
$(\mathtt{x\ after\ e2}) = \lambda t.\ 3$

These successive denotations for x are indicated graphically in figure 2. The dotted part of the first trajectory for x indicates that part of the predicted future for x that was altered by the occurrence of event e2. Figure 3 shows a trajectory for x obtained by stitching these two denotations together. Figure 3 cheats slightly, because it doesn't make clear what the value of x should be at the instant e2 occurs. We have the means available to make

locations may drift at some specified rate from the global clock.

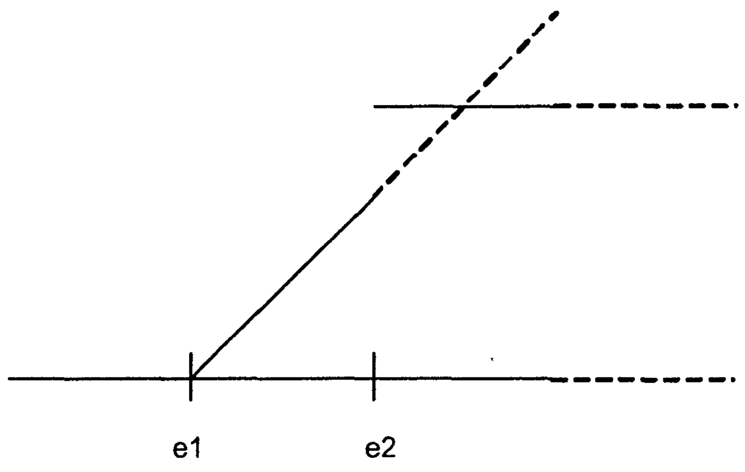

Figure 2: Successive denotations of x

the choice in whatever way is most convenient—for example, to make the trajectory left continuous or right continuous, or even to say that the "real" position of x at that time is some nondeterministic value between 2 and 3. In any case, choosing to summarize this information in a single "trajectory" artifact, whose definition involves an arbitrary choice, doesn't cause any information to be lost.

Finally, as is customary in modeling hybrid systems, we introduce an axiom that rules out "Zeno" behaviors in which infinitely many events occur within a finite span of time: for any location i and any time $t \geq 0$, there is a latest event at i whose local time is not greater than t.

4.2 Hybrid message automata

The clauses and basic proof rules for hybrid message automata, which generalize the clauses and proof rules for message automata to apply to the case of continuous variables, are described in section 4.2.1. A hybrid message automaton in which all variables are discrete is equivalent to an ordinary message automaton. Section 4.5 applies the logic of hybrid message automata to a distributed generalization of Fischer's protocol, a mutual exclusion algorithm that relies on timing assumptions.

These hybrid message automata *cannot* realize a timeliness requirement

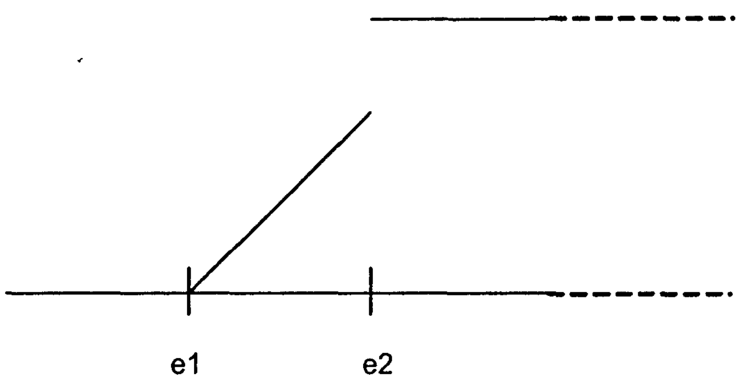

Figure 3: A trajectory for x

such as "perform task A every 100 milliseconds." That is as it should be, since one can't say anything about such a requirement in the abstract, without knowing what resources are available. We factor out timeliness requirements by using them as explicit *hypotheses*. Thus, our methods provide the logical glue between high-level requirements and low-level timing requirements that can be addressed with standard scheduling techniques. This strategy is described in section 4.2.3. Carrying it out is a subject for Phase II research.

4.2.1 Clauses and logical rules

Every variable will be declared as being either discrete or continuous. Updating a variable assigns a trajectory to it (i.e., a function of time), and we require that a discrete variable must be assigned a constant trajectory. That requirement modifies the clauses and rules for initialization and effects, and modifies the definition of compatibility between clauses.

The requirement that actions be chosen "fairly" must be slightly modified. The intuitive idea of fairness is that an action cannot become permanently enabled but permanently ignored (not taken). In an ordinary message automaton, a variable can change its value only as the result of some event; and therefore the eligibility of an action does not change between events. The technical definition of fairness for ordinary message automata relied on the fact that "nothing changes while we're not looking."

But a continuous variable, such as a clock, may change its value in between observations, and therefore the eligibility of an action may change between observations. So the technical statement of the fairness requirement must be rephrased so that "becomes permanently true" means "is true now and at *all* times in the future" rather than "is true now and true in every *observed* future state."

We have implemented the logic of hybrid message automata in NuPrl and exercised it in two ways.

First, we performed a sanity check: Every theorem about ordinary message automata should be true of hybrid message automata under the assumption that all variables are discrete. Accordingly, we redeveloped the entire theory of ordinary message automata, and its library of some 2,400 lemmas, in this new setting. The labor was considerable, because it involved changes to notions at the heart of the theory—such as the denotation of a variable and the axiom relating **when** and **after**. The NuPrl tactic mechanism, which allowed us to replay old proofs and to upgrade the auxiliary tactics they used, was essential to making this manageable.

Second, we applied it to the generalized Fischer protocol defined in section 4.5. This example includes reasoning about "delay" statements and network latency but includes no timeliness requirements.

4.2.2 Code generation for hybrid message automata

Code generation for hybrid automata is work for Phase II. Here we note one essential point. The meaning of a discrete variable is clear, since it corresponds to a program variable. But what is the computational meaning of a continuous variable? We distinguish between those continuous variables that are used as specification artifacts—e.g., to represent the trajectories of physical objects in some phase space—and those with direct computational meaning. At least one kind of continuous variable will have such a meaning: a clock or a timer.

Clocks (counting up) and timers (counting down) can be set and read. Starting clock `clk` with initial value `v` corresponds to the assignment

$$\texttt{clk} \ := \lambda t.\ t + \texttt{v}$$

That is, `clk` is "now" `v` and in future it increases in perfect synchrony with the conceptual global clock. Setting timer `tmr` to count down from `d` corresponds to assignment

$$\texttt{trm} \ := \lambda t.\ \texttt{d} - t$$

Each kind of continuous variable with direct computational meaning will have to be supported by its own idioms for code generation. (See also the discussion of virtual variables in section 4.2.3.)

4.2.3 Timeliness

As noted, we may use the logic of hybrid message automata to reduce a high-level property to an implementation that is correct under the assumption that its components satisfy some low-level timeliness hypotheses. From a logical point of view, these hypotheses can be arbitrary. However, rather than reproduce the whole of scheduling theory within NuPrl, it seems reasonable to apply standard scheduling tools to verify these low-level requirements, which means that the hypotheses must be translatable into the kinds of scheduling problems that the tools can handle.

Thus we need some stylized way to represent the hypotheses. The precise way to do that is a matter for Phase II, but as a simple first draft, consider a "deadline" clause written

```
deadline (P -> clk <= f)
```

where `P` is boolean, `clk` is a clock, and `f` is an expression of type `Time`.

Thus, a deadline clause would state the *hypothesis* that the boolean expression

```
P -> clk <= exp
```

is true at all times. The antecedent `P` is just a way specifying when we're actually interested in having the clock bound hold.

So, to require that some action a occur at least once every 100 seconds we could add clauses that do the following:

- introduce a clock variable `clk` that is affected only by action a;
- initialize `clk` to 0;
- include "`clk := \lambda t. t`" among the effects of a;

  This resets `clk` to 0 on completion of a.
- and add the hypothesis `deadline (true -> clk <= 100)`

By adding a few more clauses, we could require a to be strictly periodic with a period of 100. For common requirements such as strict periodicity we could introduce a convenient notation that adds all these clauses at once.

Since `clk` is introduced purely for sake of stating a hypothesis, code generated for action a should not implement any of these new clauses. Thus, we should introduce a class of virtual variables that can be used for specification purposes only. Simple static semantic restrictions would suffice to ensure that the virtual variables would have no effect on the visible behavior of the automaton.

4.3 Methodology

We provide special support for a certain style of program derivation, which is encapsulated in a collection of definitions and proof tactics. As already noted, a specification is determined by a property of event structures—which, formally, is a NuPrl term ϕ with the type

$$\lambda es.\ \phi(es) : \mathit{EventStructure} \rightarrow \mathit{Proposition}$$

That is, if the variable es is an event structure, $\phi(es)$ is a proposition. The NuPrl term denoting the specification that ϕ defines is written

$$\vdash_{es}\ \phi(es)$$

where the "es" is a bound variable. The English language meaning of this specification is

> There exists a realizer R that is feasible, and such that every event structure es consistent with R satisfies the proposition $\phi(es)$.

This specification is the top-level goal to be proven in NuPrl

More generally, specifications may be parameterized in an arbitrarily complex way. For example, section 4.4 derives a leader election protocol in a ring of size n, for any natural number n. Thus, the form of that specification is actually

$$\forall n : \mathit{Nat}\ \vdash_{es}\ \phi(es, n)$$

To generate an implementation for a ring of size 7, we derive (in one trivial step) the specification

$$\vdash_{es}\ \phi(es, 7)$$

and apply the code generator to that.

A user begins by invoking the tactic

```
Realizer 'foo'
```

on the top-level goal, where `foo` may be any string, chosen by the user, to name the realizer that will be constructed. This tactic reduces the user's task to proving three subgoals, which can be informally stated as follows:

1. `foo` is a realizer
2. `foo` is feasible
3. assuming (1), (2), and that es is consistent with `foo`, prove the proposition $\phi(es)$

The user ignores subgoals (1) and (2), because those will be addressed automatically. The user's task is to to figure out what `foo` should be, by applying NuPrl reasoning to decompose (3) top-down, until all its leaves are instances of base-case formulas (as described in section 2.3).

Having reached this point, the user clicks on the `CreateRealizer` button, which checks the user's work and, if the user hasn't erred, completes the proof automatically. In slightly, more detail:

- It checks that the leaves of the proof tree really are instances of base-case formulas.
- From these base-case formulas, it creates a definition for the realizer `foo`.
- Using that definition of `foo`, it completes the proofs of (1), (2), and (3) automatically.

  If the leaves of the proof truly are base-case formulas, the proofs of (1) and (3) are guaranteed to succeed. The proof of (2) is essentially a check for static semantic errors, such as defining an action that contradicts a frame clause.[5]

Given any theorem of the form $\vdash_{es}\ \phi(es)$, however it has been proven, clicking on the `GenerateJava` button will generate a Java implementation (provided that the NuPrl types of the state variables and messages are restricted to those that we know how to translate into Java).

[5]Strictly speaking, the feasibility condition is not decidable because NuPrl type checking and subtype checking is not decidable. However, by slightly strengthening the feasibility condition we could make it decidable without ruling out any programs likely to be practically interesting.

4.4 Example: Leader election in a ring

"Leader election" is a simple example of mutual exclusion: some set A of agents is to run a protocol that chooses exactly one of them. It's customary to call the chosen agent the "leader."

To make this formal, we declare an event kind `leader` and define "the agent at location a is chosen as the leader" to mean that an event of kind `leader` occurs at location a. Given a finite set A of locations, there must be exactly one $a \in A$ for which an event of kind `leader` occurs at a. Note that this specification contains both a liveness requirement (there must exists an event of kind `leader` at some location a in the set A) and a safety requirement (events of kind `leader` may not occur at two different locations in A).

In the formal language, locations are members of the type Id. It is convenient to model a finite set of locations by a list containing no repeats. Then the formal specification is

$$\begin{array}{l} \forall L:\ Id\ \texttt{List}\ \texttt{no\_repeats}(L) \Rightarrow \\ \qquad \models_{es} \exists ldr \in L\ (\exists e@_{es} ldr.kind_{es}(e) = \texttt{leader}) \\ \qquad\qquad\qquad \wedge \\ \qquad\qquad\qquad (\forall i \in L.\forall e@_{es} i.\ kind_{es}(e) = \texttt{leader} \Rightarrow i = ldr) \end{array}$$

This specification is parameterized by a list of agents (so, in particular, the result we prove will apply to any number of agents). The way in which we parameterize results is not restricted by the template mechanisms of any programming language. All the types and resources of NuPrl are available.[6]

Different formal proofs of this logical statement will lead to different realizers (i.e., different protocols) and different Java code. Our formal proof showed that the list L could be made into a ring by adding suitable links, and thus the members of L could be assigned unique numbers (because L has no repeats). Every agent in L sends the following messages to its successor in the ring:

- a "vote" containing its own number;
- any "vote" message it receives containing a number that is greater than its own

[6]When the parameterization of the formal theorem is very simple, it may be desirable to generate code that uses templates in the target language. That will be investigated in Phase II.

Exactly one agent (the one with the biggest number) will receive a vote containing its own number. An agent who receives its own number in a vote message will perform an event of kind `leader`.

The proposition we prove has the form

$$\forall L \ldots \models_{es} \phi(L, es)$$

We generate code from statements of the form

$$\models_{es} \phi(es)$$

To generate code, we instantiate the parameter L by a list of the agents. The code generated for a list `["a","b","c","d","e"]` is given in appendix B.

To demonstrate the code we must make events of kind `leader` visible, so we add a requirement to the specification, stating that every event of kind sends a message to the external node `output`.

4.5 Example: A generalization of Fischer's protocol

Fischer's Protocol has been used as a benchmark for comparing methods of program analysis. We construct a somewhat artificial test by extending it to the distributed case and use it to experiment with reasoning about real time in hybrid event systems.

4.5.1 Fischer's Protocol

Fischer's Protocol is a mutual exclusion algorithm that relies on timing assumptions. Its standard formulation relies on all processes sharing a variable—which is called `turn` in the pseudo-code below.

Let the processes be numbered $1, \ldots, N$. Process i executes:

```
loop
   wait for turn = 0;
   L: turn := i;
   delay b;
   if turn = i then
       enter critical section;
       turn := 0;
   endif;
end loop;
```

When turn is 0 the critical resource is available. In order to claim the resource, a process writes its own id into turn, then waits for at least time b and reads turn again. If turn still contains its id, the process enters its critical section and, on concluding it, resets turn to 0. (We assume that both reading turn and setting turn are atomic actions.)

Without timing assumptions this doesn't guarantee exclusion. Suppose that the value of turn is 0 and the actions of process 1 and process 2 are interleaved as follows:

process 1 reads `turn`
process 2 reads `turn`
process 1 executes `turn := 1`
process 1 rereads `turn`, finding value 1
process 1 enters its critical section
process 2 executes `turn := 2`
process 2 rereads `turn`, finding value 2
process 2 enters its critical section

Now, both processes are in their critical sections at the same time.

Mutual exclusion will be guaranteed if the execution time of "`turn := i`" is strictly less than the length of delay `b`. Notice that this protocol makes essential use of the shared variable `turn`, something unavailable in a distributed system.

4.5.2 A distributed version

We have formulated a distributed version of this protocol, which proves to be somewhat tricky. We describe it informally. A node can be in one of five states, whose intuitive meanings are given in figure 4.

`free`	believe that the shared resource is free
`taken`	know that the resource is taken by some other node
`try`	attempting to acquire the resource
`contend`	contending with others for the resource
`mine`	possessing the resource

Figure 4: The states of a node

The transitions between states are shown in Figure 5.

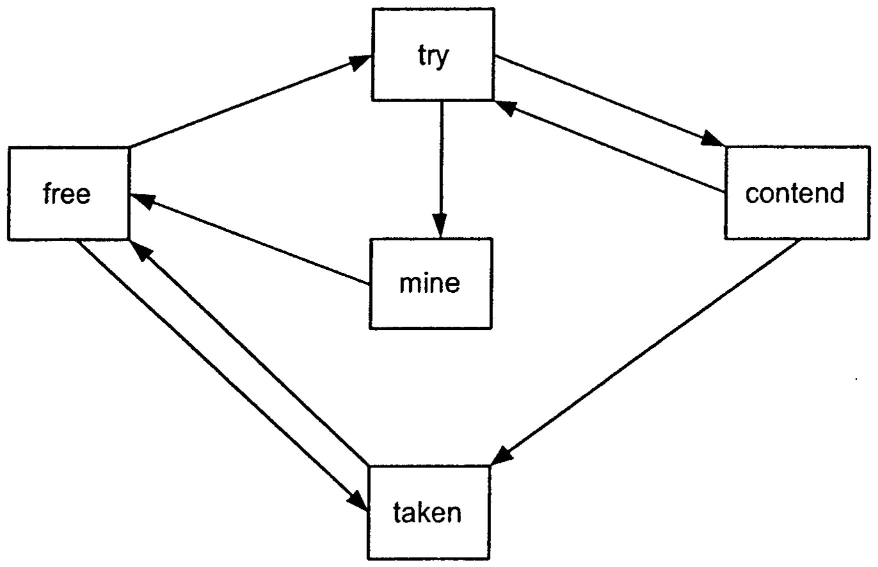

Figure 5: Distributed Fischer's Protocol

Certain transitions are guarded by conditions (including timing requirements) and certain transitions result in the sending of broadcast messages. We assume that

- these broadcasts are delivered reliably, with a latency of at most d
- no nodes fail

The preconditions and effects of those transitions in Figure 6.

This is not meant to be a practical protocol, since the underlying assumptions are very strong. The main goal was to provide a nontrivial example on which to exercise the model of hybrid message automata.

4.5.3 Lessons learned

We carried out a derivation for this protocol far enough to show that our formalism is rich enough to handle it:

- stated the specification formally;

`free` $\longrightarrow$ `try`
Requires: in state `free` for at least d
Broadcast: `"trying"`

`free` $\longrightarrow$ `taken`
Trigger: receive `"taken"`

`taken` $\longrightarrow$ `free`
Trigger: receive `"free"` message

`try` $\longrightarrow$ `mine`
Trigger: in state `try` for $2d$, without receiving `"trying"`
Broadcast: `"taken"`

`mine` $\longrightarrow$ `free`
Broadcast: `"free"`

`try` $\longrightarrow$ `contend`
Trigger: receive `"trying"` within $2d$ of entering `try`
Chooses: set `r` to positive random number

`contend` $\longrightarrow$ `taken`
Trigger: receive `"trying"` between $2d$ and $2d + \texttt{r}$ after entering `contend`

`contend` $\longrightarrow$ `try`
Trigger: `"trying"` not received between $2d$ and $2d + \texttt{r}$ after entering `contend`

Figure 6: Preconditions and effects of transitions

- defined a system invariant and proved formally (in NuPrl) that it implies the specification;
- defined a set of constraints and showed, by an informal pencil-and-paper proof, that they are realizable and collectively imply the invariant.

  Parts of the pencil-and-paper proof were carried out fully formally, which lead to significant improvements in our machinery for supporting proofs (described below) and pointed out directions for future development.

The formal work resulted in: improving `MaAuto`, our basic tactic for doing routine reasoning about message automata; developing a tactic for deciding rational constraints; introducing a logic for call-by-value; and suggesting the high-level technique of "reasoning about knowledge" to simplify the proof.

The invariant The key invariant says that execution of the protocol can be thought of as occurring in a series of "rounds," each of which contains "slices." Every round begins with all nodes in state `free` and ends with one node in state `mine` and all others in state `taken`. One possible run of the protocol, with three nodes, is shown in figure 7. The arrows represent messages. All the events on the same horizontal line constitute a slice. The invariant states that any two events in the same slice occur less than δ time units apart and that, within a single round, the sequence of states occurring at any one node must match the regular expression

```
free  (try; contend)* (taken | mine)
```

Rational constraints The proofs required a great deal of reasoning by cases, in which it was "obvious" that many of the cases would be impossible—e.g., a case in which all the following inequalities would have to hold for the timing of events e and e':

$$\begin{aligned} \texttt{time(e)} + \delta &< \texttt{time(e')} \\ \texttt{time(e')} + \delta &< \texttt{time(e)} \\ 0 &< \delta \end{aligned}$$

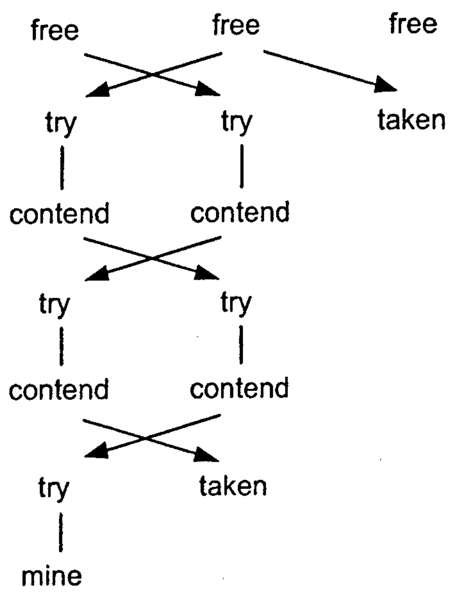

Figure 7: One round of the protocol

We developed a provably correct tactic that, given any such set of constraints (all with rational coefficients), would return either a particular solution, or guarantee that none could exist.

Call-by-value The logic of NuPrl is defined, in part, by a computation system: a collection of rules for reducing terms. Typically, these rules are *lazy*—reductions will be delayed as long as possible, so that they won't be performed unless they are required. Thus, for example, the expression "`(\lambda x.true)y`" will persist until it occurs in a context such as "`(if \lambda x.true)y then A else B`"—at which point the function application will reduce to `true` and the conditional will reduce to `A`.

In certain situations lazy evaluation turns out to be costly. So we introduced a term

```
call_by_value(y, x.B[x])
```

that represents an "eager" evaluation of `(\lambda x. B[x])y`. It means "evaluate y, and if it has a value, substitute that value for the x in `B[x]`."

The tactic we have defined for reasoning about timing inequalities did some bookkeeping in a 2-dimensional table that was represented as a list of lists. The first version of the tactic did table lookups lazily, leading to a proliferation of large terms that symbolically denoted the results of table lookup. The result was very large, inefficiently handled, formulas. It proved much more efficient to do lookups using call-by-value. A vivid example of this effect can be seen by programming the Fibonacci function: an implementation that takes exponentially many steps if additions are evaluated lazily becomes linear if additions are evaluated with call-by-value.

Reasoning about knowledge Prof. Joe Halpern, of Cornell University, noted that the key invariant we formulated for the generalized Fischer protocol is an instance of "ε-common knowledge," as defined in [20]. The relevance of reasoning about knowledge is also suggested by the terminology used in figure 4. Reasoning abstractly about knowledge, and then instantiating to our particular case, could greatly simplify tedious details. A Cornell PhD student is currently working on representing such reasoning within our event system formalism.

5 Results and Discussion

The goal of our work is to provide automated support for the development of correct distributed and/or real-time systems. Developers will work at a high level of abstraction, specifying requirements in a declarative way and reasoning about them in a domain specific logic. Using the "proofs as programs" paradigm, a developer shows, with powerful automated support, that some high-level requirement *can* be implemented—and that demonstration itself defines an abstract implementation, called a *(hybrid) message automaton*, which we automatically extract and from which we generate the code for an implementation.

5.1 Phase I

Building on previous work that extended the "proofs as programs" approach from sequential to distributed systems, Phase I research achieved the following results:

We implemented a prototype code generator to translate message automata into distributed Java programs and demonstrated it on a leader election protocol. There was of course no doubt that we could extract message automata from proofs and could generate programs from the automata. Building the prototype accomplished two things: First, it created an infrastructure for extracting message automata from proofs and manipulating them. In Phase II, this infrastructure can be generalized to handle *hybrid* message automata (which model real-time behavior) and can be retargeted, if desired, to generate code in languages other than Java. Second, we identified what parameterization will be required to build a practical code generation tool. For example, many implementation choices—such as whether senders "push" messages or receivers "pull" them—concern performance, not correctness. The prototype builds in a default choice, but a Phase II implementation of the code generator will allow the user to make those decisions from a high-level menu of choices.

We extended the logic of distributed systems to incorporate real time and time-varying variables, and extended message automata to a corresponding notion of hybrid message automata. We demonstrated this logic by applying it to distributed mutual exclusion (using a version of Fischer's protocol). Using this example (and others) we have begun the work of developing proof methods that scale, including automated tactics that will accomplish much of the routine work of proving. A good library of tactics is essential to making our methods practical.

5.2 Phase II

We summarize a number of technical tasks remaining for Phase II. All these tasks have been described earlier in this report.

5.2.1 The formal model

The formal model uses a standard mathematical trick allowing it to model dynamic behavior by an unbounded static model. All possible nodes and all possible connections among them exist already, and by using appropriate "create," "destroy," and "connect" actions we can *simulate* dynamic behavior. A code generator cannot implement this procedure. We need to incorporate into our logic a set of idioms that hides the static model and refers directly to dynamic behavior, from which code generation can take its cues.

5.2.2 Proof methods

Section 4.5.3 noted several ways in which we have made proving more efficient (and therefore more scaleable). Such improvements typically respond to problems discovered while doing examples. Phase I work has already allowed us to identify additional goals:

- Generalize ESP to real-time.

  Previous work has defined and implemented the ESP notation (section 2.5), which can be thought of as a way to define a realizer using something like a regular expression.

- Allow the introduction of new proof rules in which reasoning steps are carried out by invoking decision procedures, or semi-decision procedures, external to NuPrl; dependencies upon such rules must be carefully accounted for.

- Improve the efficiency of code generation by exploiting symmetries. (For example, if all nodes of a network are essentially running the same protocol, we want to avoid duplicating work by generating the code of each node from scratch.)

The major addition required in Phase II will be support for reasoning about timeliness. By themselves, hybrid message automata cannot guarantee timeliness constraints, such as "task A is executed once every 100 milliseconds." We have proposed to partition the reasoning: using such timeliness constraints as *hypotheses*, thereby reducing the high-level requirements to this set of hypothesized constraints, and then applying standard scheduling tools to satisfy those constraints.

In the abstract, these hypotheses can be arbitrary; but if we want to translate them into standard scheduling problems we must express them in some restricted and stylized form. We must design that notation and make the connection to scheduling tools. Phase I work has proposed a first draft.

5.2.3 Code generation

We must extend code generation from message automata to hybrid message automata. Where the formal model (and therefore the correctness guarantees) allows a variety of implementation choices, we must give a developer the ability to tell the code generator which choice to make. These choices will involve communication protocols, scheduling, security, etc.

6 Bibliography

References

[1] Uri Abraham. On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science*, 149:257–298, 1995.

[2] Uri Abraham. *Models for Concurrency*, volume 11 of *Algebra, Logic and Applications Series*. Gordon and Breach, 1999.

[3] Uri Abraham, Shlomi Dolev, Ted Herman, and Irit Koll. Self-stabilizing ℓ-exclusion. *Theoretical Computer Science*, 266:653–692, 2001.

[4] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of Lecture Notes in Artificial Intelligence, pages 170-176. Springer Verlag, 2000.

[5] Stuart Allen, Mark Bickford, Robert Constable, Rich Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, special issue on Mathematics Assistance Systems, expected 2005.

[6] Stuart Allen, Mark Bickford, Robert Constable, et al. FDL: A prototype formal digital library, May 2002. http://www.nuprl.org/html/FDLProject/02cucs-fdl.html.

[7] Myla Archer and Constance Heitmeyer. Mechanical verification of timed automata: A case study. Technical report, Naval Research Laboratory, Washington, DC 20375, May 19, 1997. A shorter version of this report was presented at RTAS '96, Boston, MA, June 10–13, 1996.

[8] Andrew Barber, Philippa Gardner, Masahito Hasegawa, and Gordon D. Plotkin. From action calculi to linear logic. In Mogens Nielsen and Wolfgang Thomas, editors, *Computer Science Logic, 11th International Workshop, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 78–97. Springer, 1998.

[9] Ricky W. Butler and George B. Finelli The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. IEEE Transactions on Software Engineering, 19(1), pages 3–12, January 1993.

[10] Levente Buttyán and Jean-Pierre Hubaux. Report on a working session on security in wireless ad hoc networks, 2002.

[11] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[12] Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In Jeannette Wing, Jim Woodcock, and J. Davies, editors, *FM99: The World Congress in Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589, 1999.

[13] Robert L. Constable, Stuart Allen, Mark Bickford, James Caldwell, Jason Hickey, and Christoph Kreitz. Steps Toward a World Wide Digital Library of Formal Algorithmic Knowledge *MURI Review*, volume 1. 2003.

[14] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons from branching time temporal logic. In *Proc. Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer–Verlag, 1982.

[15] J. Douceur, A. Adya, J. Benaloh, W. Bolosky, and G. Yuval. A secure directory service based on exclusive encryption. In *18th ACSAC*, December 2002.

[16] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[17] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A program refinement framework supporting reasoning about knowledge and time. In Jerzy Tiuryn, editor, *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2000)*, pages 114–129, Berlin/New York, 1998. Springer-Verlag.

[18] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A refinement theory that supports reasoning about knowledge and time for synchronous agents. In Robert Nieuwenhuis and Andrei Voronkov, editors, *8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 125–141. Springer-Verlag, December 2001.

[19] http://www.nuprl.org/FDLproject/

[20] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. Massachusetts Institute of Technology, 1995.

[21] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

[22] Joseph Y. Halpern. A note on knowledge-based programs and specifications. *Distributed Computing*, 13(3):145–153, 2000.

[23] Joseph Y. Halpern and Ronald Fagin. Modeling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–177, 1989.

[24] Joseph Y. Halpern and Richard A. Shore. Reasoning about common knowledge with infinitely many agents. In *Proceedings of the 14th*

IEEE Symposium on Logic in Computer Science, pages 384–393, 1999.

[25] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer-Verlag, New York, 2003.

[26] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[27] Isabelle home page. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[28] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In *Proceedings of Seventh International Coference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 2001.

[29] K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software–Practice and Experience*, 24(7):643–658, 1994.

[30] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In A. Arora, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems, Austin, TX*, pages 33–40. IEEE Computer Society Press, 1999.

[31] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, 21(7):558–65, 1978.

[32] Leslie Lamport. Hybrid systems in TLA+. In Grossman, Nerode, Ravn, and Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, 1993.

[33] Leslie Lamport. The temporal logic of actions. 16(3):872–923, 1994.

[34] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Boston, 2003.

[35] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[36] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, Berlin, 1992.

[37] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, Berlin, 1995.

[38] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. and Syst.*, 6(1):68–93, 1984.

[39] R. Milner. Action structures and the π-calculus. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20–August 1, 1993, NATO Series F*, pages 219–280. Springer, Berlin, 1994.

[40] Robin Milner. *Communication and Concurrency.* Prentice-Hall, London, 1989.

[41] Robin Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[42] Jayadev Misra. *A Discipline of Multiprogramming.* 2001.

[43] Lawrence C. Paulson. Mechanizing UNITY in isabelle. *ACM Transactions on Computational Logic*, 1999.

[44] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of Thirty-first IEEE Symposium on Foundations of Computer Science*, pages 746–757, 1990.

[45] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods: PROCOMET'98*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

[46] Fred B. Schneider. *On Concurrent Programming.* Springer-Verlag, New York, 1997.

[47] Gerard J. Holzmann *The Spin model checker: primer and reference manual.* Addison-Wesley, 2004.

[48] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor, *Computer Aided Verification, Proceedings of the 7th International Conference*, volume 939 of *Lecture Notes in Computer Science*, pages 267–292. Springer-Verlag, 1995.

[49] G. Winskel. *Events in Computation.* PhD thesis, University of Edinburgh, 1980.

[50] G. Winskel. An introduction to event structures. In J. W. de Bakker et al., editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 345 in Lecture Notes in Computer Science, pages 364–397. Springer, 1989.

[51] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.

[52] Job Zwiers, Willem P. de Roever, and Peter van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In *ICALP 1985*, pages 509–519, 1985.

A Design of the generated code

The design of the code, which is suitable for a distributed system whose nodes and links are known at runtime, is straightforward. We generate a package for each link, and one for each node (with a slight qualification to be discussed later).

The package associated with a link provides

- classes defining the messages that may be sent on the link (along with their accompanying tags);
- interfaces `Sender` and `Receiver` defining the operations available to a node sending and receiving on the link;
- implementations of `Sender` and `Receiver`.

Some nodes represent external actors—devices, or software not under our control. For demonstration purposes, we generate simple virtual models of such external actors—they allow inputs to be entered through a gui and write outputs to a terminal. To limit the amount of time spent programming guis, the prototype makes the simplifying restriction that each external actor is attached to only one link.

The package associated with an "internal" node provides

- a class Node that implements a message automaton;

  The constructor for a Node has parameters: a lnk.Receiver object for each link lnk on which it receives messages, and a lnk.Sender object for each link lnk to which it sends messages.

- a Main program that creates the node and starts it running.

It is straightforward to replace a virtual actor with a real one or with a simulator: create appropriate Receiver and Sender objects for the links to which the actor is attached, and revise Main so that it passes in these new objects as parameters to the constructors for nodes attached to the link. (This phrasing assumes that any external actor will be wrapped up inside a Java class.)

The remaining sections describe the code in slightly more detail.

A.1 The package for a link

Each link package contains a class Tag that supplies symbolic names for the tags of all the messages that can be sent on the link, and also supplies the name None to mean "no tag." A tag is an abstract message header. Its underlying representation will be an integer.

For writing examples in this section, suppose that the tags for messages send on lnk are x, y, and z; and suppose that messages with tag x have type T_x, etc.

In this case, the package lnk exports the following public classes and interfaces (with inheritance shown in the usual way by indentation):

```
Classes                     Interfaces

    Tag                     Receiver
    Tagged                  Sender
       Tagged_x
       Tagged_y
       Tagged_z
    send
    rcv
```

A.1.1 Tags and tagged values

Class Tag merely supplies symbolic names for the tags. If no one intends to read this generated code, there is no need for this class. But it makes the generated code a bit clearer.

```
public class Tag{
    public static final int None = 0;
    public static final int x = 1;
    public static final int y = 2;
    public static final int z = 3;
}
```

The value None is a convenience meaning "no tag." So the particular tags used in messages on link lnk have unique global names of the form lnk.Tag.x, etc.

The values sent on the link are tagged values, which are objects of subclasses of class Tagged.

```
abstract public class Tagged implements Serializable{
    int tag;
    public Tagged(int i){
       tag = i;
    }
}
```

By making the Tagged class Serializable, we automatically get behind-the-scenes operations that marshal elements of its subclasses when we send them on a socket.

Objects representing tagged values are created in the obvious way.

```
public class Tagged_x extends Tagged{
   T_x val;
   public Tagged_hello(T_x i){
      super(Tag.x);
      val = i;
   }
}
```

This looks a little odd, since the class exports no methods for accessing the members of a Tagged_x. The members, of course, have package-wide visibility, and we can access them that way. We will never change the value of a Tagged_x object once it's been created.

A.1.2 The interface Sender

The Sender interface provides, unsurprisingly, methods for sending Tagged values.

```
public interface Sender {

    public void send_x(T_x d);
    public void send_y(T_y d);
    public void send_z(T_z d);

    public void close();
}
```

It would perhaps look a bit more "OO" for this Interface to provide a single method

```
    public void send(Tagged t);
```

and add send() and get() methods to type Tagged, so that transmitting a value v with tag x might be done by saying

```
    (new Tagged_x(v)).send();
```

Since we are generating code, and since we are not reusing any of the classes we generate or subclassing them, the distinction seems largely cosmetic. Adding methods to descendants of Tagged would also tie the definitions of Tagged_x, etc., to the implementation of Send and Receive, so might lead to a little more complexity.

The close() method—and the corresponding is_closed() query provided by Receiver—are conveniences. Using these methods, a node may discover that its incoming links are closed and then close its outgoing links. They suffice to allow our demonstrations to terminate gracefully.

A.1.3 The interface Receiver

The prototype expects a Receiver to accept messages as they arrive and buffer them. It provides methods giving access to and information about the message buffer:

- is_msg() returns the tag of the top message in the buffer, if the buffer is nonempty, and otherwise returns the value None.
- get_x() should be called only when the buffer is nonempty and its top element has tag x, and then it removes the message and returns its value.
- set_notice(n) supplies a "notification" object n that the receiver can use to signal the arrival of a new message. A node can use these notifications to avoid busy waiting.
- The purpose of is_closed() has been described in section A.1.3.

```
public interface Receiver {

    public int is_msg();
    public T_x get_x();
    public T_y get_y();
    public T_z get_z();

    public void set_notice(note.Note n);
    public void is_closed();

}
```

A.1.4 Default implementation of Sender

We describe send, an implementation of interface Sender for a link between two internal nodes. (We don't bother to describe the implementation of sending to the screen, which is our virtual output node.)

```
public class send{
   ...
      /* Sets up connection */
   public send() throws InterruptedException;
   public close(); /* Closes connection */

      /* Send messages on the link. */
   public void send_x(T_x v);
```

```
    public void send_y(T_y v);
    public void send_x(T_z v);
}
```

Setting up and closing the connection are not part of the formal model, which starts with a collection of already existing nodes and links. We implement sending on a link between two internal nodes as sending on and receiving from a socket. The send operation creates a client socket, through which it attempts to connect to an appropriate server. If send can't find the host at the given address, it fails and quits. If it finds the host but the server socket does not yet exist, send catches the resulting IOException, goes to sleep for a prescribed time, and then tries again, repeatedly. Once successfully connected, it creates an ObjectOutputStream from that connection. The operations send_x, etc., merely write to that output stream.

The IP address of the receiving end of the socket is supplied as a parameter to the code generator. The code generator chooses the port number for this connection: it assigns different ports to all incoming links at a node, using port numbers 1024 and above. (Since an application may want to use some of these higher port numbers for other purposes, the code generator should also be supplied with a list of port numbers to avoid. We have not implemented that.) The operations of the corresponding rcv class will listen on that port.

We have not tried to accommodate specifying and responding to all the errors that might arise when a connection is being set up or when a connection breaks.

A.1.5 Default implementation of Receiver link

We describe rcv, the implementation of interface Receiver, whether the source of the link is a node sending messages on a socket or a virtual node modeled as a gui. Comments will indicate the differences.

Members of rcv A rcv class creates

- buffer, to hold arriving messages (so it's actually a buffer of Tagged elements);
- a Note object note with sleep and wake methods, which will allow us to implement the top-level loop of a node without using a busy wait;

- a new thread receive, that listens on the appropriate port and places arriving messages in buffer, signaling with note.wake() when a new message arrives;
- is_notice, a boolean that will be set when its note object has been created;
- and, if messages are arriving on a socket,
  - listener, a ServerSocket that listens on the appropriate port
  - oin, an input stream from listener

The code generator chooses a port number and hardwires the same port number into the code for the corresponding sender.

```
Vector buffer;
note.Note notice;
boolean closed;

/* Objects needed if messages come from a socket. */

ServerSocket listener;
...
ObjectInputStream oin;
Thread receiver;
```

The constructor If the link is a socket, we create a new thread receiver, which listens on the socket and puts incoming messages into buffer. If the link gets its messages from a gui, then the sender itself directly places messages in the buffer. (Compare the two different fill_buffer operations.)

```
public rcv(note.Note n){
   try {
      closed = false;
      buffer = new Vector();

      /* Initializations of socket-related objects. */
```

```
          listener = new ServerSocket(...);
          ...
          oin = new ObjectInputStream(...);
          receiver = new Thread(
             new Runnable(){public void run(){fill_buffer();}}
          );
         receiver.start();
      catch {...}
      }
   }
```

fill_buffer The fill_buffer operation adds a message to buffer and signals a wakeup. This method synchronizes on buffer, because different threads can access buffer.

If the input comes from a gui, rcv will make fill_buffer visible to send, which can use it to add a message directly to the buffer.

```
    void fill_buffer(Tagged x){
        synchronized(buffer){buffer.add(x);}
        notice.wake();
    }
```

If the input comes from a socket, fill_buffer is a loop that runs in its own thread and adds messages to buffer as they arrive.

```
    private void fill_buffer(){
        Tagged tmp;
        try {
            while(true){
                tmp = (Tagged)oin.readObject();
                synchronized(buffer){buffer.add(tmp);}
                while (!notice){
                  Thread.currentThread().sleep(1000);
                }       /* This busy wait occurs only once. */
                notice.wake();
            }
        }
        catch (EOFException e){
           closed = true;
```

```
            notice.wake();
        }
        catch (Exception e){...}
    }
```

The not very elegant inner loop forces the thread running fill_buffer to wait until the notice object has been created. Once it has been created, the inner loop is a no-op.

The other methods of rcv The set_notice method supplies a value for the notice member and sets is_notice to true. The methods is_msg(), and get_*() are described in section A.1.3. The implementations are trivial and the same in both cases. It necessary that is_msg() and the get_*() methods synchronize on buffer).

A.2 The package for a node

The package for each node defines a class Node that is independent of how its links are implemented, and a package Main that instantiates Node with implementations of the link ends to which it is attached.

A node will consist of two threads, each executing a loop. One loop, encapsulated in a Receiver object, will place incoming messages in a buffer; and the other, which we call "main loop," will repeatedly check for eligible actions and execute them. We try to avoid busy waiting in the main loop when there is nothing for it to do.

Suppose that a node has

- incoming links in1 and in1
- outgoing link out1
- state variables s1 and s2 of types T1 and T2

A.2.1 Data members

For every abstract state variable s of the message automaton we declare two member variables, s and s_next. Each action of the node will proceed by computing the next value for s and storing it in s_next; sending any required messages (whose values may depend of the original value of s, which is as yet unchanged); and then copying s_next into s. In many cases it will be possible to optimize the copying step away and modify s directly.

The other member data consists of

- instances of the link ends to which the node is attached
- control variables for receiving notifications and keeping track of progress
- a variable for each action, to store the value it returns

```
public class Node implements Runnable {
// state variables -- current value and next value
   T1 s1, next_s1;
   T2 s2, next_s2;

// ends of links

   out1.Sender to_out1;
   in1.Receiver from_in1;
   in2.Receiver from_in2;

// action values
   V1 act1_value;
   V1 act2_value;
   ...

// control variables
   boolean progress;
   note.Note notice;

...
}
```

A.2.2 Constructing a Node

The parameters to Node are implementations of the appropriate link ends. Most of the task of setting up the links will be done in the Main program, when it creates Sender and Receiver objects; but the node must supply the Receivers with a notification object.

```
// constructor
Node(in1.Receiver from1, in2.Receiver from2, out1.Sender to1)
    throws InterruptedException {
// initialize state variables
   ...  // Default initial values defined by formal model

   progress = false;

// set up link ends
   notice = new note.Note();
   from1.set_notice(notice);
   from2.set_notice(notice);
}
```

There's one subtlety in setting up the link ends—namely, doing it in the right order. That might depend on the implementation of the link ends, so the hard-coded order in which this is currently done is a design flaw.

To see the problem, consider the case in which links are sockets and, as in our default implementations, the attempt to set up a client socket at the input end of a link persists until the receiving end has set up the corresponding server socket. So, if we have links

```
lnk1: M -> N            lnk2: N -> M
```

and `M` tries to set up the sending end of `lnk1`, while `N` tries to set up the sending end of `lnk2`, the setup process will deadlock. The code generator constructs a local ordering at each node that will avoid such deadlocks.

A.2.3 Auxiliary operations

The auxiliary operations compute

- The precondition and value of each action: `check_act1` returns `false` if there is no value that, in the current state, satisfies the precondition for `act1`; otherwise, returns `true` and updates `act1_value` appropriately.
- The effect of each action: `do_act1` performs action 1 and sets `progress` to `true`.
- The response to an incoming message on each link (depending on its tag):

- check_in1() examines the buffer for link 1 and, if it finds a message with tag x, invokes do_in1_x
- do_in1_x performs the response to an incoming message on link 1 with tag x and sets progress to true.

```
// Look for a solution to precondition of an action

boolean check_act1() {...}
boolean check_act2(){...}
...

// Check incoming links and, if appropriate, respond

void check_in1(){
   switch(from1.is_msg()){
      case in1.Tag.None: break;
      case in1.Tag.x: {
         do_in1_x(from1.get_x())
         break;
      }
      case: ...
   }
 }

void check_in2(){...}

//  Actions (triggered by acts or by messages)

void do_act1(){
// compute next value for state variables
// do any sends
// update current values to next values
progress = true;
}

void do_act2(){...}
...
```

```
void do_in1_x(T_x val){
// compute next value for state variables
// do any sends
// update current values to next values
progress = true;
}

...

// Bookkeeping for link status

// Check whether all in-links are closed
boolean allInLinksClosed() {
   boolean answer;
   answer = true;
   answer = answer & from1.is_closed() & from2.is_closed();
   return answer;
}

//close all out-links
public void closeAllOutLinks() {
   to1.close();
}
```

A.2.4 The main loop

Abstractly, a message automaton loop repeatedly chooses some eligible action or some incoming message and performs the action or responds to the message. The formal model only requires only that these choices be "fair"—it cannot happen that an action or response is permanently enabled but never performed. Any schedule fair in this sense will be correct. Our default implementation is essentially round-robin.

In our default implementation, each iteration of the main loop checks, in succession, the precondition of each local action and, if the precondition is true, executes it. (Note that these executions may change the state and therefore change the status of subsequent preconditions.) Then it checks the message buffer for each incoming link and, for each buffer that is nonempty, responds to the topmost message.

We complicate this in two ways: First, we wish to avoid busy waiting when there's nothing for the automaton to do. Second, we wish to provide a graceful way of terminating simple demonstrations. (Think of this graceful exit as a stand-in for shutting down a node in a more complex situation.)

We say that an iteration of the loop makes progress if and only if it executed at least one local action or response. A variable `progress` keeps track of this. The local state of an automaton can be affected only by an iteration that makes progress. If an iteration fails to make progress, there is nothing further for the automaton to do until a new message arrives. (Lack of progress means, in particular, that the message buffer is empty.) So the loop suspends itself, to be awakened when a new message arrives. If an iteration does not make progress and the node learns that, in addition, all its incoming links have closed—so that no more messages will arrive—it will close its outgoing links and terminate.

Under the hood, suspension is done with Java's `wait()` statement and awakening with a corresponding `notifyAll()`. Both the suspending/awakening mechanism and the mechanism for shutting down may, if implemented naively, result in race conditions. Consider, for example, the following sequence of events:

Main loop	Receiver thread
no progress	
	new message
	`notifyAll()`
`wait()`	
	no more messages

The `notifyAll()` occurs before the main loop suspends itself by invoking `wait()`. So, if no more messages arrive, the main loop is never awakened from its `wait()` statement, and therefore never responds to the final message.

To avoid this sort of problem we provide the appropriate coordination with the synchronizing operations `consider_sleep()` and `wake()` of the `Note` object that the two threads share. In particular, the `progress` variable is a member of that object.

Section C contains a SPIN model [47] of our code and describes how we use SPIN to verify (to a moral certainty) that our code is correct.

```
   // main loop
   public void run(){
      boolean done;
      done = false;
      while(!done){
         notice.progress = false;
         //  check each local action
         if (check_act1()) { do_act1();}
         if (check_act2()) { do_act2();}
         ...
         // check each incoming link
         check_in1();
         check_in2();
         if (! notice.progress & allInLinksClosed()) { done = true; }
         else { notice.consider_sleep(); }
         }
     }// end while
     closeAllOutLinks();
   }//end run
```

A.3 The package note

A `Note` object packages up our little protocol for suspending and awakening.

```
public class Note {
    public boolean progress = false;
    public synchronized void consider_sleep() {
        if (! progress) {try {
            wait ();
        }
        catch (InterruptedException e) {}
        }
    }

 public synchronized void wake () {
     progress = true;
     notifyAll ();
 }
}
```

B Code for leader election in a ring

Since the code is completely symmetric, we show only the packages for node a, for the link aXbX1 from node a to node b, and the link aXoutputX1 from a to external output. We also show the trivial, handwritten configuration file that assigns nodes to processors.

B.1 Configuration file

In this case, we execute the five nodes on two machines, and refer to the machines by their network names rather than their IP addresses.

```
// comments and white space are ignored
 a = "mojave"
 c = "mojave"
 e = "mojave"
 b = "nuprl1"
 d = "nuprl1"

// if destinations are on same host then ports must be distinct
 a,b,"1" = 1024
 c,d,"1" = 1025
 e,a,"1" = 1024
 b,c,"1" = 1025
 d,e,"1" = 1026
```

B.2 Package a

B.2.1 Class Main

```
package a;
public class Main {
public Main(){}
public static void main(String[] args) {
Node nd;
try {
aXbX1.send e1 = new aXbX1.send();
aXoutputX1.send e2 = new aXoutputX1.send();
eXaX1.rcv e3 = new eXaX1.rcv();
nd = new Node(e1,e2,e3);
Thread node = new Thread(nd);
node.start();
} catch (InterruptedException e) {
}
}// end main
}//end Main
```

B.2.2 Class Node

```
package a;
import java.util.Vector;
import java.lang.Integer;
import note.*;
import javax.swing.*;
public class Node implements Runnable {
// state variables -- current and next
boolean done1, next_done1;
int me, next_me;
boolean trigger, next_trigger;
// ends of links
aXbX1.Sender toXbX1;
aXoutputX1.Sender toXoutputX1;
eXaX1.Receiver fromXeX1;
// action values
boolean send_DASH_meXvalue;
boolean leaderXvalue;
note.Note notice;
// constructor
Node(aXbX1.Sender XtoXbX1, aXoutputX1.Sender XtoXoutputX1,
eXaX1.Receiver XfromXeX1) throws InterruptedException {
// initialize state variables
done1 = false;
next_done1 = false;
me = 0;
next_me = 0;
trigger = false;
next_trigger = false;
```

```
// set up link ends
notice = new note.Note();
toXbX1 = XtoXbX1;
toXoutputX1 = XtoXoutputX1;
fromXeX1 = XfromXeX1;
fromXeX1.set_notice(notice);
notice.progress = false;
}

boolean checkXsend_DASH_me() {
if (
(done1 ? false : true)
) {
send_DASH_meXvalue = true;
return true;}
else {return false;}
}
boolean checkXleader() {
if (
(trigger ? true : false)
) {
leaderXvalue = true;
return true;}
else {return false;}
}
void checkXfromXeX1(){
switch(fromXeX1.is_msg()){
case eXaX1.Tag.None: break;
case eXaX1.Tag.vote: {
doXfromXeX1Xvote(fromXeX1.get_vote());
break; }
} //end switch
}//end check

void doXsend_DASH_me() {
// update nexts
next_done1 = true;
// do any sends
toXbX1.send_vote(me);
// assign currents
```

```
done1 = next_done1;
notice.progress = true;
}

void doXfromXeX1Xvote(int val) {
// update nexts
next_trigger = ((me == val) ? true : trigger);
// do any sends
if (
(me < val)
) {
toXbX1.send_vote(val);
} else {
}
// assign currents
trigger = next_trigger;
notice.progress = true;
}

void doXleader() {
// update nexts
// do any sends
toXoutputX1.send_leader(me);
// assign currents
notice.progress = true;
}

//check if all in-links are closed
boolean allInLinksClosed() {
boolean answer;
answer = true;
answer = answer & fromXeX1.is_closed();
return answer;
}//end allInLnksClosed
//close all out-links
public void closeAllOutLinks() {
toXbX1.close();
toXoutputX1.close();
}//end closeAllOutLinks
// main loop
```

```
public void run(){
boolean done;
done = false;
 while(!done){
notice.progress = false;
//  check each local action
if (checkXsend_DASH_me()) { doXsend_DASH_me();}
if (checkXleader()) { doXleader();}
// check each incoming link
checkXfromXeX1();
if (! notice.progress & allInLinksClosed()) {
System.out.println("a: all in-links closed & no progess possible.");
System.out.println("a exiting.");
   done = true; }
 else {
System.out.println("a sleeping.");
notice.consider_sleep();
System.out.println("a awake.");
}// else
}// end while
closeAllOutLinks();
}//end run

}// end Node
```

B.3 Package aXbX1

B.3.1 Interface Receiver

```
package aXbX1;
public interface Receiver {
public void set_notice(note.Note n);
public int is_msg();
public int get_vote();
public boolean is_closed();
}
```

B.3.2 Class rcv

```
package aXbX1;
```

```
import java.net.*;
import java.io.*;
import java.util.Vector;
import note.*;
import config.*;
public class rcv implements Receiver {
ServerSocket listener;
Socket client;
InputStream in;
ObjectInputStream oin;
Vector buffer;
Thread receiver;
note.Note notice;
boolean closed;
boolean is_notice = false;
public rcv(){
try {
closed = false;
listener = new ServerSocket(config.Config.port("a","b","1"));
client = listener.accept();
in = client.getInputStream();
oin = new ObjectInputStream(in);
buffer = new Vector();
receiver = new Thread(
new Runnable(){public void run(){fill_buffer();}}
);
receiver.start();
System.out.println("aXbX1 receiver connected.");
}
catch (UnknownHostException e){
System.out.println("aXbX1.rcv:"+ e + "Can't find host.");
}
catch (IOException e){
System.out.println("aXbX1.rcv:"+ e + "Error connecting to host.");
}
}

public void set_notice(note.Note n){
   notice = n;
   is_notice = true;
```

```
 }

private void fill_buffer(){
Tagged tmp;
try {
while(true){
tmp = (Tagged)oin.readObject();
synchronized(buffer){buffer.add(tmp);}
System.out.println("aXbX1 calling wake.");
while(!is_notice) {Thread.currentThread().sleep(1000);}
notice.wake();
}
}
catch (EOFException e){
System.out.println("Link aXbX1 closed.");
closed = true;
notice.wake();
}
catch (Exception e){
System.out.println("fillbuffer@aXbX1:"+ e +
"Error reading from host or queueing input.");
}
}
public int is_msg(){
int tag;
synchronized(buffer){
if (!buffer.isEmpty())
{ tag = ((Tagged)buffer.firstElement()).tag;}
else { tag = Tag.None;}
}
return tag;
}
public int get_vote(){
int
val;
synchronized(buffer){
val = ((Tagged_vote)buffer.firstElement()).val;
  buffer.remove(0);
}
return val;
```

```
}
public boolean is_closed() {
   return closed;
}
}
```

B.3.3 Interface Sender

```
package aXbX1;
public interface Sender {
public void send_vote(int d);
public void close();
}
```

B.3.4 Class send

```
package aXbX1;
import java.net.*;
import java.io.*;
import java.lang.Thread.*;
import config.*;
public class send implements Sender{
Socket sock;
OutputStream out;
ObjectOutputStream oout;
InetAddress inaddr;
InetSocketAddress addr;
public send() throws InterruptedException {
boolean done = false;
while (!done) {
try {
inaddr = InetAddress.getByName(config.Config.location("b"));
addr = new InetSocketAddress(inaddr,config.Config.port("a","b","1"));
sock = new Socket();
sock.connect(addr,0);
out = sock.getOutputStream();
oout = new ObjectOutputStream(out);
done = true;
System.out.println("aXbX1 sender connected.");
}
catch (UnknownHostException e){
System.out.println("aXbX1.send:"+ e + "Can't find host.");
done = true;
}
catch (IOException e){
Thread.currentThread().sleep(1000);
}
}
}
public void send_vote(int d){
try {
System.out.println("aXbX1 sending " + d + " with tag vote.");
oout.writeObject(new Tagged_vote(d));
}
catch (IOException e){
System.out.println("aXbX1:"+ e + "Error in output stream.");
}
catch (Exception e ){
System.out.println("aXbX1:"+ e + "Error: This can't happen!");
}
}
public void close(){
try {sock.close();}
catch (IOException e) { }
}//end close
}//end send class
```

B.3.5 Class Tag

```
package aXbX1;
public class Tag{
public static final int None = 0;
public static final int vote = 1;
public Tag(){}
}
```

B.3.6 Class Tagged

```
package aXbX1;
import java.io.*;
```

```
abstract public class Tagged implements Serializable{
int tag;
public Tagged(int i){
tag = i;
}
}
```

B.3.7 Class Tagged_vote

```
package aXbX1;
import java.io.*;
public class Tagged_vote extends Tagged{
int val;
public Tagged_vote(int i){
super(Tag.vote);
val = i;
}
}
```

B.4 Package aXoutputX1

B.4.1 Interface Sender

```
package aXoutputX1;
public interface Sender {
public void send_leader(int d);
public void close();
}
```

B.4.2 Class send

```
package aXoutputX1;
public class send implements Sender{
public void send_leader(int v)
{System.out.println("Output1: leader(" + v + ")");}
public void close(){System.out.println("Output1 closed.");}
public send() {
}
}
```

B.4.3 Interface Receiver

```
package aXoutputX1;
public interface Receiver {
public void set_notice(note.Note n);
public int is_msg();
public int get_leader();
public boolean is_closed();
}
```

B.4.4 Class Tag

```
package aXoutputX1;
public class Tag{
public static final int None = 0;
public static final int leader = 1;
public Tag(){}
}
```

B.4.5 Class Tagged

```
package aXoutputX1;
import java.io.*;
abstract public class Tagged implements Serializable{
int tag;
public Tagged(int i){
tag = i;
}
}
```

B.4.6 Class Tagged_leader

```
package aXoutputX1;
import java.io.*;
public class Tagged_leader extends Tagged{
int val;
public Tagged_leader(int i){
super(Tag.leader);
val = i;
}
}
```

C A verification of the sleep/wake/terminate protocol

We model the essence of our sleep/wake/terminate protocol in Promela, the modeling language of SPIN, and use the SPIN model-checker to verify its key property. We want to show that problems like the one described in section A.2.4 cannot occur—that is, the behavior on every link is *correct* in the sense that all messages received on that link are responded to.

We simplify the model by introducing several sound abstractions:

1. The only relevant program variables are `progress`, `done` (which determines when the node will terminate its main loop), and `buffer` (in which received messages are stored).

2. We model only one link. As explained in (5), that is sufficient.

3. The content of messages is irrelevant. All that's relevant is the size of the buffer. Therefore, we model the response to a message simply as the act of removing the message from the buffer.

4. We model the operation determining whether all incoming links are closed by a boolean variable, `allInLinksClosed`, which may be set nondeterministically to `true` at any point *after* it is guaranteed that no more messages will arrive on the link we model.

5. Local actions, or responses to messages on other links, can only affect the variable `progress`, and can do that only by setting it to `true`. We model those effects by statements that, at appropriate times, nondeterministically choose whether or not to execute `progress=true`.

The model is parameterized by an integer N, the total number of messages received on the link. Here lies the formal gap in our argument: we rely on the intuitive truth that, if there is a timing problem, it can be exhibited by an execution sequence containing not terribly many messages. (And, in an act of overkill, we verify the model for values of N up to 50.)

Execution starts with `buffer` empty. The property we want to show is:

> Every possible execution will eventually reach a state in which
> the message buffer is empty and remains so permanently.

That means that all incoming messages will be processed.

Thus, execution may set `allInLinksClosed` to `true` at any time after N messages have been inserted into `buffer`—or may never set it to `true`. This covers all possibilities: that the closing of incoming links occurs, but the discovery of that fact is delayed arbitrarily; that it occurs but is never discovered; or that some links remain open after the link we that we are modeling finishes.

The following SPIN model is straightforward, and should be easily understood by anyone who knows Promela. We model the `wait()` and `notifyAll()` actions as follows:

- The declaration of the process representing the main thread says that it is active "`provided ( !waiting )`"
- The main thread executes `wait()` by setting `waiting` to `true`.
- The receiver thread executes `notifyAll()` by setting `waiting` to `false`.

The synchronized methods `consider_sleep` and `wake` explicitly acquire and release a lock—remembering that a thread gives up its locks if it executes a `wake()`.

```
#define true 1
#define false 0

/* The definition of 'empty_buffer' is used to state the specification
   we will check:

            <>[]empty_buffer

   Strictly speaking, we're checking that the negation of this is a
   valid never-claim. */

#define empty_buffer (len(buffer)==0)

/* Variables modeling operating system state. */

bit waiting, locked;
```

```
/* Macros modeling operating system actions. */

#define wait          waiting=true
#define notifyAll     waiting=false

#define get_lock      locked=true
#define release_lock  locked=false

/* The two synchronized methods, consider_sleep and wake. */

#define consider_sleep                                       \
    d_step{! locked -> get_lock };                           \
    if                                                       \
    :: ! progress ->  d_step{ release_lock; wait };          \
                      d_step{! locked -> get_lock }          \
    :: else -> skip                                          \
    fi;                                                      \
    release_lock

#define wake                                                 \
    d_step{! locked -> get_lock };                           \
    progress = true;                                         \
    notifyAll;                                               \
    release_lock


/* Counts down from N, to keep track of the messages sent.
   After N messages, the sender will stop. */

int counter;

/* The remaining variables occur in the program. */

int progress, done, allInLinksClosed;

/* The message buffer maintained by the receiver.
   The value of 'N' is supplied on SPIN's command line. */
```

```
chan buffer = [N] of {bit};

proctype main() provided ( ! waiting ) {
   done = false;
   do
   :: progress = false;

      /* Some local action or a response on another
         link might result in progress. */
      if
      :: progress = true
      :: skip
      fi;

      /* Now check for messages on this link. */
      if
      :: atomic{buffer?[_] -> buffer?_};   /* SPIN idiom */
         progress = true
      :: else -> skip
      fi;

      /* Some other local action or a response on another
         link might result in progress. */
      if
      :: progress = true
      :: skip
      fi;

      /* The termination check */
      if
      :: ! progress && allInLinksClosed -> done = true
      :: else -> skip
      fi;

      consider_sleep
   od
}
```

```
proctype receiver(){
   do
   :: counter == 0 ->  break
   :: else -> buffer!0; counter--; wake
   od;     /* It doesn't matter what value is
              put in the buffer. */

   /* Once all messages are sent on this link,
      it may or may not happen that all in links
      become closed. */
   if
   :: allInLinksClosed = true
   :: skip
   fi
}

init{ waiting=false; counter=N; allInLinksClosed=false;
      run main(); run receiver() }
```